VISIT…



Version Control with Subversion

For Subversion 1.4

(Compiled from r2866)

Ben Collins-Sussman
Brian W. Fitzpatrick
C. Michael Pilato

Version Control with Subversion: For Subversion 1.4: (Compiled from r2866)

by Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato

Published (TBA)

Copyright © 2002, 2003, 2004, 2005, 2006, 2007 Ben Collins-SussmanBrian W. FitzpatrickC. Michael Pilato

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit http://creativecommons.org/licenses/by/2.0/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Table of Contents

Foreword xi
Preface xiii
- Audience xiii
- How to Read this Book xiv
- Conventions Used in This Book xv
  - Typographic Conventions xv
  - Icons xv
- Organization of This Book xv
- This Book is Free xvi
- Acknowledgments xvii
  - From Ben Collins-Sussman xvii
  - From Brian W. Fitzpatrick xvii
  - From C. Michael Pilato xviii
- What is Subversion? xviii
  - Subversion's History xix
  - Subversion's Features xix
  - Subversion's Architecture xxi
  - Subversion's Components xxii

1. Fundamental Concepts 1
- The Repository 1
- Versioning Models 2
  - The Problem of File-Sharing 2
  - The Lock-Modify-Unlock Solution 3
  - The Copy-Modify-Merge Solution 5
- Subversion in Action 7
  - Subversion Repository URLs 7
  - Working Copies 8
  - Revisions 11
  - How Working Copies Track the Repository 13
  - Mixed Revision Working Copies 14
- Summary 15

2. Basic Usage 16
- Help! 16
- Getting Data into your Repository 16
  - svn import 16
  - Recommended repository layout 17
- Initial Checkout 17
  - Disabling Password Caching 19
  - Authenticating as a Different User 19
- Basic Work Cycle 20
  - Update Your Working Copy 20
  - Make Changes to Your Working Copy 21
  - Examine Your Changes 22
  - Undoing Working Changes 26
  - Resolve Conflicts (Merging Others' Changes) 26
  - Commit Your Changes 30
- Examining History 31
  - Generating a list of historical changes 32
  - Examining the details of historical changes 33
  - Browsing the repository 35
  - Fetching older repository snapshots 36

Sometimes You Just Need to Clean Up 37
Summary 37
3. Advanced Topics 38
Revision Specifiers 38
Revision Keywords 38
Revision Dates 39
Properties 40
Why Properties? 41
Manipulating Properties 42
Properties and the Subversion Workflow 45
Automatic Property Setting 46
File Portability 47
File Content Type 47
File Executability 48
End-of-Line Character Sequences 49
Ignoring Unversioned Items 50
Keyword Substitution 53
Locking 56
Creating locks 58
Discovering locks 60
Breaking and stealing locks 61
Lock Communication 63
Externals Definitions 64
Peg and Operative Revisions 67
Network Model 71
Requests and Responses 71
Client Credentials Caching 71
4. Branching and Merging 75
What's a Branch? 75
Using Branches 76
Creating a Branch 77
Working with Your Branch 80
The Key Concepts Behind Branches 82
Copying Changes Between Branches 82
Copying Specific Changes 82
The Key Concept Behind Merging 85
Best Practices for Merging 86
Common Use-Cases 89
Merging a Whole Branch to Another 89
Undoing Changes 91
Resurrecting Deleted Items 93
Common Branching Patterns 94
Traversing Branches 96
Tags 98
Creating a Simple Tag 98
Creating a Complex Tag 98
Branch Maintenance 99
Repository Layout 100
Data Lifetimes 100
Vendor branches 101
General Vendor Branch Management Procedure 102
svn_load_dirs.pl 104
Summary 105
5. Repository Administration 107
The Subversion Repository, Defined 107
Strategies for Repository Deployment 108
Planning Your Repository Organization 108

Deciding Where and How to Host Your Repository 111
Choosing a Data Store 111
Creating and Configuring Your Repository 115
Creating the Repository 115
Implementing Repository Hooks 116
Berkeley DB Configuration 117
Repository Maintenance 118
An Administrator's Toolkit 118
Commit Log Message Correction 122
Managing Disk Space 122
Berkeley DB Recovery 125
Migrating Repository Data Elsewhere 126
Filtering Repository History 130
Repository Replication 133
Repository Backup 138
Summary 140
6. Server Configuration 141
Overview 141
Choosing a Server Configuration 142
The **svnserve** Server 142
svnserve over SSH 143
The Apache HTTP Server 143
Recommendations 144
svnserve, a custom server 144
Invoking the Server 144
Built-in authentication and authorization 147
Tunneling over SSH 149
SSH configuration tricks 151
httpd, the Apache HTTP server 152
Prerequisites 153
Basic Apache Configuration 154
Authentication Options 155
Authorization Options 159
Extra Goodies 163
Path-Based Authorization 166
Supporting Multiple Repository Access Methods 169
7. Customizing Your Subversion Experience 172
Runtime Configuration Area 172
Configuration Area Layout 172
Configuration and the Windows Registry 173
Configuration Options 174
Localization 178
Understanding locales 178
Subversion's use of locales 179
Using External Differencing Tools 180
External diff 181
External diff3 182
8. Embedding Subversion 184
Layered Library Design 184
Repository Layer 185
Repository Access Layer 189
Client Layer 189
Inside the Working Copy Administration Area 191
The Entries File 191
Pristine Copies and Property Files 191
Using the APIs 192
The Apache Portable Runtime Library 192

URL and Path Requirements 193
Using Languages Other than C and C++ 194
Code Samples 194
9. Subversion Complete Reference 200
The Subversion Command Line Client: **svn** 200
svn Options 200
svn Subcommands 204
svnadmin 268
svnadmin Options 268
svnadmin Subcommands 269
svnlook 286
svnlook Options 286
svnlook Subcommands 287
svnsync 303
svnsync Options 303
svnsync Subcommands 304
svnserve 307
svnserve Options 308
svnversion 309
mod_dav_svn 311
Subversion properties 313
Versioned Properties 313
Unversioned Properties 313
Repository Hooks 314
A. Subversion Quick-Start Guide 324
Installing Subversion 324
High-speed Tutorial 325
B. Subversion for CVS Users 328
Revision Numbers Are Different Now 328
Directory Versions 328
More Disconnected Operations 329
Distinction Between Status and Update 329
Status 330
Update 331
Branches and Tags 331
Metadata Properties 331
Conflict Resolution 331
Binary Files and Translation 332
Versioned Modules 332
Authentication 332
Converting a Repository from CVS to Subversion 333
C. WebDAV and Autoversioning 334
What is WebDAV? 334
Autoversioning 335
Client Interoperability 336
Standalone WebDAV applications 338
File-explorer WebDAV extensions 339
WebDAV filesystem implementation 340
D. Third Party Tools 342
E. Copyright 343
Index 348

List of Figures

1. Subversion's Architecture xxi
1.1. A typical client/server system 1
1.2. The problem to avoid 2
1.3. The lock-modify-unlock solution 3
1.4. The copy-modify-merge solution 5
1.5. The copy-modify-merge solution (continued) 5
1.6. The repository's filesystem 9
1.7. The repository 11
4.1. Branches of development 75
4.2. Starting repository layout 76
4.3. Repository with new copy 78
4.4. The branching of one file's history 80
8.1. Files and directories in two dimensions 187
8.2. Versioning time—the third dimension! 187

List of Tables

1.1. Repository Access URLs 10
5.1. Repository Data Store Comparison 111
6.1. Comparison of Subversion Server Options 141
C.1. Common WebDAV Clients 336

List of Examples

5.1. txn-info.sh (Reporting Outstanding Transactions) 123
5.2. Mirror repository's pre-revprop-change hook script 135
5.3. Mirror repository's start-commit hook script 135
6.1. A sample configuration for anonymous access. 161
6.2. A sample configuration for authenticated access. 161
6.3. A sample configuration for mixed authenticated/anonymous access. 161
6.4. Disabling path checks altogether 162
7.1. Sample Registration Entries (.reg) File. 173
7.2. diffwrap.sh 181
7.3. diffwrap.bat 182
7.4. diff3wrap.sh 182
7.5. diff3wrap.bat 183
8.1. Using the Repository Layer 194
8.2. Using the Repository Layer with Python 196
8.3. A Python Status Crawler 198

Foreword

Karl Fogel
Chicago, March 14, 2004

A bad Frequently Asked Questions (FAQ) sheet is one that is composed not of the questions people actually asked, but of the questions the FAQ's author *wished* people had asked. Perhaps you've seen the type before:

> Q: How can I use Glorbosoft XYZ to maximize team productivity?
>
> A: Many of our customers want to know how they can maximize productivity through our patented office groupware innovations. The answer is simple: first, click on the "`File`" menu, scroll down to "`Increase Productivity`", then…

The problem with such FAQs is that they are not, in a literal sense, FAQs at all. No one ever called the tech support line and asked, "How can we maximize productivity?". Rather, people asked highly specific questions, like, "How can we change the calendaring system to send reminders two days in advance instead of one?" and so on. But it's a lot easier to make up imaginary Frequently Asked Questions than it is to discover the real ones. Compiling a true FAQ sheet requires a sustained, organized effort: over the lifetime of the software, incoming questions must be tracked, responses monitored, and all gathered into a coherent, searchable whole that reflects the collective experience of users in the wild. It calls for the patient, observant attitude of a field naturalist. No grand hypothesizing, no visionary pronouncements here—open eyes and accurate note-taking are what's needed most.

What I love about this book is that it grew out of just such a process, and shows it on every page. It is the direct result of the authors' encounters with users. It began with Ben Collins-Sussman's observation that people were asking the same basic questions over and over on the Subversion mailing lists: What are the standard workflows to use with Subversion? Do branches and tags work the same way as in other version control systems? How can I find out who made a particular change?

Frustrated at seeing the same questions day after day, Ben worked intensely over a month in the summer of 2002 to write *The Subversion Handbook*, a sixty page manual that covered all the basics of using Subversion. The manual made no pretense of being complete, but it was distributed with Subversion and got users over that initial hump in the learning curve. When O'Reilly and Associates decided to publish a full-length Subversion book, the path of least resistance was obvious: just expand the Subversion handbook.

The three co-authors of the new book were thus presented with an unusual opportunity. Officially, their task was to write a book top-down, starting from a table of contents and an initial draft. But they also had access to a steady stream—indeed, an uncontrollable geyser—of bottom-up source material. Subversion was already in the hands of thousands of early adopters, and those users were giving tons of feedback, not only about Subversion, but about its existing documentation.

During the entire time they wrote this book, Ben, Mike, and Brian haunted the Subversion mailing lists and chat rooms incessantly, carefully noting the problems users were having in real-life situations. Monitoring such feedback was part of their job descriptions at CollabNet anyway, and it gave them a huge advantage when they set out to document Subversion. The book they produced is grounded firmly in the bedrock of experience, not in the shifting sands of wishful thinking; it combines the best aspects of user manual and FAQ sheet. This duality might not be noticeable on a first reading. Taken in order, front to back, the book is simply a

straightforward description of a piece of software. There's the overview, the obligatory guided tour, the chapter on administrative configuration, some advanced topics, and of course a command reference and troubleshooting guide. Only when you come back to it later, seeking the solution to some specific problem, does its authenticity shine out: the telling details that can only result from encounters with the unexpected, the examples honed from genuine use cases, and most of all the sensitivity to the user's needs and the user's point of view.

Of course, no one can promise that this book will answer every question you have about Subversion. Sometimes, the precision with which it anticipates your questions will seem eerily telepathic; yet occasionally, you will stumble into a hole in the community's knowledge, and come away empty-handed. When this happens, the best thing you can do is email `<users@subversion.tigris.org>` and present your problem. The authors are still there, still watching, and they include not just the three listed on the cover, but many others who contributed corrections and original material. From the community's point of view, solving your problem is merely a pleasant side effect of a much larger project—namely, slowly adjusting this book, and ultimately Subversion itself, to more closely match the way people actually use it. They are eager to hear from you not merely because they can help you, but because you can help them. With Subversion as with all active free software projects, *you are not alone*.

Let this book be your first companion.

Preface

> “It is important not to let the perfect become the enemy of the good, even when you can agree on what perfect is. Doubly so when you can't. As unpleasant as it is to be trapped by past mistakes, you can't make any progress by being afraid of your own shadow during design.”
>
> —Greg Hudson

In the world of open-source software, the Concurrent Versions System (CVS) was the tool of choice for version control for many years. And rightly so. CVS was open-source software itself, and its non-restrictive modus operandi and support for networked operation allowed dozens of geographically dispersed programmers to share their work. It fit the collaborative nature of the open-source world very well. CVS and its semi-chaotic development model have since become cornerstones of open-source culture.

But CVS was not without its flaws, and simply fixing those flaws promised to be an enormous effort. Enter Subversion. Designed to be a successor to CVS, Subversion's originators set out to win the hearts of CVS users in two ways—by creating an open-source system with a design (and “look and feel”) similar to CVS, and by attempting to avoid most of CVS's noticeable flaws. While the result isn't necessarily the next great evolution in version control design, Subversion *is* very powerful, very usable, and very flexible. And for the most part, almost all newly-started open-source projects now choose Subversion instead of CVS.

This book is written to document the 1.4 series of the Subversion version control system. We have made every attempt to be thorough in our coverage. However, Subversion has a thriving and energetic development community, so there are already a number of features and improvements planned for future versions of Subversion that may change some of the commands and specific notes in this book.

Audience

This book is written for computer-literate folk who want to use Subversion to manage their data. While Subversion runs on a number of different operating systems, its primary user interface is command-line based. That command-line tool (**svn**), and some auxiliary programs, are the focus of this book.

For consistency, the examples in this book assume the reader is using a Unix-like operating system and is relatively comfortable with Unix and command-line interfaces. That said, the **svn** program also runs on non-Unix platforms like Microsoft Windows. With a few minor exceptions, such as the use of backward slashes (`\`) instead of forward slashes (`/`) for path separators, the input to and output from this tool when run on Windows are identical to its Unix counterpart.

Most readers are probably programmers or system administrators who need to track changes to source code. This is the most common use for Subversion, and therefore it is the scenario underlying all of the book's examples. But Subversion can be used to manage changes to any sort of information—images, music, databases, documentation, and so on. To Subversion, all data is just data.

While this book is written with the assumption that the reader has never used a version control system, we've also tried to make it easy for users of CVS (and other systems) to make a painless leap into Subversion. Special sidebars may mention other version control systems from time to time, and a special appendix summarizes many of the differences between CVS and Subversion.

Note also that the source code examples used throughout the book are only examples. While they will compile with the proper compiler incantations, they are intended to illustrate a particular scenario, not necessarily serve as examples of good programming style or practices.

How to Read this Book

Technical books always face a certain dilemma: whether to cater to *top-down* or *bottom-up* learners. A top-down learner prefers to read or skim documentation, getting a large overview of how the system works; only then does she actually start using the software. A bottom-learner is a “learn by doing” person, someone who just wants to dive into the software and figure it out as she goes, referring to book sections when necessary. Most books tend to be written for one type of person or the other, and this book is undoubtedly biased towards top-down learners. (And if you're actually reading this section, you're probably already a top-down learner yourself!) However, if you're a bottom-up person, don't despair. While the book may be laid out as a broad survey of Subversion topics, the contents of each section tends to be heavy with specific examples that you can try-by-doing. For the impatient folks who just want to get going, you can jump right to Appendix A, *Subversion Quick-Start Guide*.

Regardless of your learning style, this book aims to be useful to people of widely different backgrounds—from people with no previous experience in version control to experienced system administrators. Depending on your own background, certain chapters may be more or less important to you. The following can be considered a “recommended reading list” for various types of readers:

Experienced System Administrators
: The assumption here is that you've probably used version control before before, and are dying to get a Subversion server up and running ASAP. Chapter 5, *Repository Administration* and Chapter 6, *Server Configuration* will show you how to create your first repository and make it available over the network. After that's done, Chapter 2, *Basic Usage* and Appendix B, *Subversion for CVS Users* are the fastest routes to learning the Subversion client.

New users
: Your administrator has probably set up Subversion already, and you need to learn how to use the client. If you've never used a version control system, then Chapter 1, *Fundamental Concepts* is a vital introduction to the ideas behind version control. Chapter 2, *Basic Usage* is a guided tour of the Subversion client.

Advanced users
: Whether you're a user or administrator, eventually your project will grow larger. You're going to want to learn how to do more advanced things with Subversion, such as how to use branches and perform merges (Chapter 4, *Branching and Merging*), how to use Subversion's property support (Chapter 3, *Advanced Topics*), how to configure runtime options (Chapter 7, *Customizing Your Subversion Experience*), and other things. These chapters aren't critical at first, but be sure to read them once you're comfortable with the basics.

Developers
: Presumably, you're already familiar with Subversion, and now want to either extend it or build new software on top of its many APIs. Chapter 8, *Embedding Subversion* is just for you.

The book ends with reference material—Chapter 9, *Subversion Complete Reference* is a reference guide for all Subversion commands, and the appendices cover a number of useful topics. These are the chapters you're mostly likely to come back to after you've finished the book.

Conventions Used in This Book

This section covers the various conventions used in this book.

Typographic Conventions

Constant width
Used for commands, command output, and options

`Constant width italic`
Used for replaceable items in code and text

`Italic`
Used for file and directory names

Icons



This icon designates a note relating to the surrounding text.



This icon designates a helpful tip relating to the surrounding text.



This icon designates a warning relating to the surrounding text.

Organization of This Book

The chapters that follow and their contents are listed here:

Preface
Covers the history of Subversion as well as its features, architecture, and components.

Chapter 1, *Fundamental Concepts*
Explains the basics of version control and different versioning models, along with Subversion's repository, working copies, and revisions.

Chapter 2, *Basic Usage*
Walks you through a day in the life of a Subversion user. It demonstrates how to use a Subversion client to obtain, modify, and commit data.

Chapter 3, *Advanced Topics*
Covers more complex features that regular users will eventually come into contact with, such as versioned metadata, file locking, and peg revisions.

Chapter 4, *Branching and Merging*
Discusses branches, merges, and tagging, including best practices for branching and merging, common use cases, how to undo changes, and how to easily swing from one branch to the next.

Chapter 5, *Repository Administration*
: Describes the basics of the Subversion repository, how to create, configure, and maintain a repository, and the tools you can use to do all of this.

Chapter 6, *Server Configuration*
: Explains how to configure your Subversion server and different ways to access your repository: HTTP, the svn protocol, and local disk access. It also covers the details of authentication, authorization and anonymous access.

Chapter 7, *Customizing Your Subversion Experience*
: Explores the Subversion client configuration files, the handling of internationalized text, and how to make external tools cooperate with Subversion.

Chapter 8, *Embedding Subversion*
: Describes the internals of Subversion, the Subversion filesystem, and the working copy administrative areas from a programmer's point of view. Demonstrates how to use the public APIs to write a program that uses Subversion, and most importantly, how to contribute to the development of Subversion.

Chapter 9, *Subversion Complete Reference*
: Explains in great detail every subcommand of **svn**, **svnadmin**, and **svnlook** with plenty of examples for the whole family!

Appendix A, *Subversion Quick-Start Guide*
: For the impatient, a whirlwind explanation of how to install Subversion and start using it immediately. You have been warned.

Appendix B, *Subversion for CVS Users*
: Covers the similarities and differences between Subversion and CVS, with numerous suggestions on how to break all the bad habits you picked up from years of using CVS. Included are descriptions of Subversion revision numbers, versioned directories, offline operations, **update** vs. **status**, branches, tags, metadata, conflict resolution, and authentication.

Appendix C, *WebDAV and Autoversioning*
: Describes the details of WebDAV and DeltaV, and how you can configure your Subversion repository to be mounted read/write as a DAV share.

Appendix D, *Third Party Tools*
: Discusses tools that support or use Subversion, including alternative client programs, repository browser tools, and so on.

This Book is Free

This book started out as bits of documentation written by Subversion project developers, which were then coalesced into a single work and rewritten. As such, it has always been under a free license. (See Appendix E, *Copyright*.) In fact, the book was written in the public eye, originally as a part of Subversion project itself. This means two things:

- You will always find the latest version of this book in the book's own Subversion repository.
- You can make changes to this book and redistribute it however you wish—it's under a free license. Your only obligation is to maintain proper attribution to the original authors. Of course, rather than distribute your own private version of this book, we'd much rather you send feedback and patches to the Subversion developer community.

The online home of this book's development and most of the volunteer-driven translation efforts around it is http://svnbook.red-bean.com. There, you can find links to the latest releases and tagged versions of the book in various formats, as well as instructions for accessing the book's Subversion repository (where lives its DocBook XML source code). Feedback is welcome—encouraged, even. Please submit all comments, complaints, and patches against the book sources to <svnbook-dev@red-bean.com>.

Acknowledgments

This book would not be possible (nor very useful) if Subversion did not exist. For that, the authors would like to thank Brian Behlendorf and CollabNet for the vision to fund such a risky and ambitious new Open Source project; Jim Blandy for the original Subversion name and design—we love you, Jim; Karl Fogel for being such a good friend and a great community leader, in that order.[1]

Thanks to O'Reilly and our editors, Linda Mui and Tatiana Diaz for their patience and support.

Finally, we thank the countless people who contributed to this book with informal reviews, suggestions, and fixes: While this is undoubtedly not a complete list, this book would be incomplete and incorrect without the help of: David Anderson, Jani Averbach, Ryan Barrett, Francois Beausoleil, Jennifer Bevan, Matt Blais, Zack Brown, Martin Buchholz, Brane Cibej, John R. Daily, Peter Davis, Olivier Davy, Robert P. J. Day, Mo DeJong, Brian Denny, Joe Drew, Nick Duffek, Ben Elliston, Justin Erenkrantz, Shlomi Fish, Julian Foad, Chris Foote, Martin Furter, Dave Gilbert, Eric Gillespie, David Glasser, Matthew Gregan, Art Haas, Eric Hanchrow, Greg Hudson, Alexis Huxley, Jens B. Jorgensen, Tez Kamihira, David Kimdon, Mark Benedetto King, Andreas J. Koenig, Nuutti Kotivuori, Matt Kraai, Scott Lamb, Vincent Lefevre, Morten Ludvigsen, Paul Lussier, Bruce A. Mah, Philip Martin, Feliciano Matias, Patrick Mayweg, Gareth McCaughan, Jon Middleton, Tim Moloney, Christopher Ness, Mats Nilsson, Joe Orton, Amy Lyn Pilato, Kevin Pilch-Bisson, Dmitriy Popkov, Michael Price, Mark Proctor, Steffen Prohaska, Daniel Rall, Jack Repenning, Tobias Ringstrom, Garrett Rooney, Joel Rosdahl, Christian Sauer, Larry Shatzer, Russell Steicke, Sander Striker, Erik Sjoelund, Johan Sundstroem, John Szakmeister, Mason Thomas, Eric Wadsworth, Colin Watson, Alex Waugh, Chad Whitacre, Josef Wolf, Blair Zajac, and the entire Subversion community.

From Ben Collins-Sussman

Thanks to my wife Frances, who, for many months, got to hear, “But honey, I'm still working on the book”, rather than the usual, “But honey, I'm still doing email.” I don't know where she gets all that patience! She's my perfect counterbalance.

Thanks to my extended family and friends for their sincere encouragement, despite having no actual interest in the subject. (You know, the ones who say, “Ooh, you wrote a book?”, and then when you tell them it's a computer book, sort of glaze over.)

Thanks to all my close friends, who make me a rich, rich man. Don't look at me that way—you know who you are.

Thanks to my parents for the perfect low-level formatting, and being unbelievable role models. Thanks to my son for the opportunity to pass that on.

From Brian W. Fitzpatrick

Huge thanks to my wife Marie for being incredibly understanding, supportive, and most of all,

[1] Oh, and thanks, Karl, for being too overworked to write this book yourself.

patient. Thank you to my brother Eric who first introduced me to UNIX programming way back when. Thanks to my Mom and Grandmother for all their support, not to mention enduring a Christmas holiday where I came home and promptly buried my head in my laptop to work on the book.

To Mike and Ben: It was a pleasure working with you on the book. Heck, it's a pleasure working with you at work!

To everyone in the Subversion community and the Apache Software Foundation, thanks for having me. Not a day goes by where I don't learn something from at least one of you.

Lastly, thanks to my Grandfather who always told me that “freedom equals responsibility.” I couldn't agree more.

From C. Michael Pilato

Special thanks to Amy, my best friend and wife of nine incredible years, for her love and patient support, for putting up with the late nights, and for graciously enduring the version control processes I've imposed on her. Don't worry, Sweetheart—you'll be a TortoiseSVN wizard in no time!

Gavin, there probably aren't many words in this book that you can successfully “sound out” at this stage, but when you've finally got a handle on the written form of this crazy language we speak, I hope you're as proud of your Daddy as he is of you.

Aidan, Daddy luffoo et ope Aiduh yike contootoo as much as Aiduh yike batetball, base-ball, et bootball. [2]

Mom and Dad, thanks for your constant support and enthusiasm. Mom- and Dad-in-law, thanks for all of the same *plus* your fabulous daughter.

Hats off to Shep Kendall, through whom the world of computers was first opened to me; Ben Collins-Sussman, my tour-guide through the open-source world; Karl Fogel—you *are* my `.emacs`; Greg Stein, for oozing practical programming know-how; Brian Fitzpatrick—for sharing this writing experience with me. To the many folks from whom I am constantly picking up new knowledge—keep dropping it!

Finally, to the One who perfectly demonstrates creative excellence—thank You.

What is Subversion?

Subversion is a free/open-source version control system. That is, Subversion manages files and directories, and the changes made to them, over time. This allows you to recover older versions of your data, or examine the history of how your data changed. In this regard, many people think of a version control system as a sort of “time machine”.

Subversion can operate across networks, which allows it to be used by people on different computers. At some level, the ability for various people to modify and manage the same set of data from their respective locations fosters collaboration. Progress can occur more quickly without a single conduit through which all modifications must occur. And because the work is versioned, you need not fear that quality is the trade-off for losing that conduit—if some incor-

[2] Translation: Daddy loves you and hopes you like computers as much as you like basketball, baseball, and football. (Wasn't that obvious?)

rect change is made to the data, just undo that change.

Some version control systems are also software configuration management (SCM) systems. These systems are specifically tailored to manage trees of source code, and have many features that are specific to software development—such as natively understanding programming languages, or supplying tools for building software. Subversion, however, is not one of these systems. It is a general system that can be used to manage *any* collection of files. For you, those files might be source code—for others, anything from grocery shopping lists to digital video mixdowns and beyond.

Subversion's History

In early 2000, CollabNet, Inc. (http://www.collab.net) began seeking developers to write a replacement for CVS. CollabNet offers a collaboration software suite called CollabNet Enterprise Edition (CEE) of which one component is version control. Although CEE used CVS as its initial version control system, CVS's limitations were obvious from the beginning, and CollabNet knew it would eventually have to find something better. Unfortunately, CVS had become the de facto standard in the open source world largely because there *wasn't* anything better, at least not under a free license. So CollabNet determined to write a new version control system from scratch, retaining the basic ideas of CVS, but without the bugs and misfeatures.

In February 2000, they contacted Karl Fogel, the author of *Open Source Development with CVS* (Coriolis, 1999), and asked if he'd like to work on this new project. Coincidentally, at the time Karl was already discussing a design for a new version control system with his friend Jim Blandy. In 1995, the two had started Cyclic Software, a company providing CVS support contracts, and although they later sold the business, they still used CVS every day at their jobs. Their frustration with CVS had led Jim to think carefully about better ways to manage versioned data, and he'd already come up with not only the name “Subversion”, but also with the basic design of the Subversion data store. When CollabNet called, Karl immediately agreed to work on the project, and Jim got his employer, Red Hat Software, to essentially donate him to the project for an indefinite period of time. CollabNet hired Karl and Ben Collins-Sussman, and detailed design work began in May. With the help of some well-placed prods from Brian Behlendorf and Jason Robbins of CollabNet, and Greg Stein (at the time an independent developer active in the WebDAV/DeltaV specification process), Subversion quickly attracted a community of active developers. It turned out that many people had had the same frustrating experiences with CVS, and welcomed the chance to finally do something about it.

The original design team settled on some simple goals. They didn't want to break new ground in version control methodology, they just wanted to fix CVS. They decided that Subversion would match CVS's features, and preserve the same development model, but not duplicate CVS's most obvious flaws. And although it did not need to be a drop-in replacement for CVS, it should be similar enough that any CVS user could make the switch with little effort.

After fourteen months of coding, Subversion became “self-hosting” on August 31, 2001. That is, Subversion developers stopped using CVS to manage Subversion's own source code, and started using Subversion instead.

While CollabNet started the project, and still funds a large chunk of the work (it pays the salaries of a few full-time Subversion developers), Subversion is run like most open-source projects, governed by a loose, transparent set of rules that encourage meritocracy. CollabNet's copyright license is fully compliant with the Debian Free Software Guidelines. In other words, anyone is free to download, modify, and redistribute Subversion as he pleases; no permission from CollabNet or anyone else is required.

Subversion's Features

When discussing the features that Subversion brings to the version control table, it is often helpful to speak of them in terms of how they improve upon CVS's design. If you're not familiar with CVS, you may not understand all of these features. And if you're not familiar with version control at all, your eyes may glaze over unless you first read Chapter 1, *Fundamental Concepts*, in which we provide a gentle introduction to version control.

Subversion provides:

Directory versioning
: CVS only tracks the history of individual files, but Subversion implements a “virtual” versioned filesystem that tracks changes to whole directory trees over time. Files *and* directories are versioned.

True version history
: Since CVS is limited to file versioning, operations such as copies and renames—which might happen to files, but which are really changes to the contents of some containing directory—aren't supported in CVS. Additionally, in CVS you cannot replace a versioned file with some new thing of the same name without the new item inheriting the history of the old—perhaps completely unrelated—file. With Subversion, you can add, delete, copy, and rename both files and directories. And every newly added file begins with a fresh, clean history all its own.

Atomic commits
: A collection of modifications either goes into the repository completely, or not at all. This allows developers to construct and commit changes as logical chunks, and prevents problems that can occur when only a portion of a set of changes is successfully sent to the repository.

Versioned metadata
: Each file and directory has a set of properties—keys and their values—associated with it. You can create and store any arbitrary key/value pairs you wish. Properties are versioned over time, just like file contents.

Choice of network layers
: Subversion has an abstracted notion of repository access, making it easy for people to implement new network mechanisms. Subversion can plug into the Apache HTTP Server as an extension module. This gives Subversion a big advantage in stability and interoperability, and instant access to existing features provided by that server—authentication, authorization, wire compression, and so on. A more lightweight, standalone Subversion server process is also available. This server speaks a custom protocol which can be easily tunneled over SSH.

Consistent data handling
: Subversion expresses file differences using a binary differencing algorithm, which works identically on both text (human-readable) and binary (human-unreadable) files. Both types of files are stored equally compressed in the repository, and differences are transmitted in both directions across the network.

Efficient branching and tagging
: The cost of branching and tagging need not be proportional to the project size. Subversion creates branches and tags by simply copying the project, using a mechanism similar to a hard-link. Thus these operations take only a very small, constant amount of time.

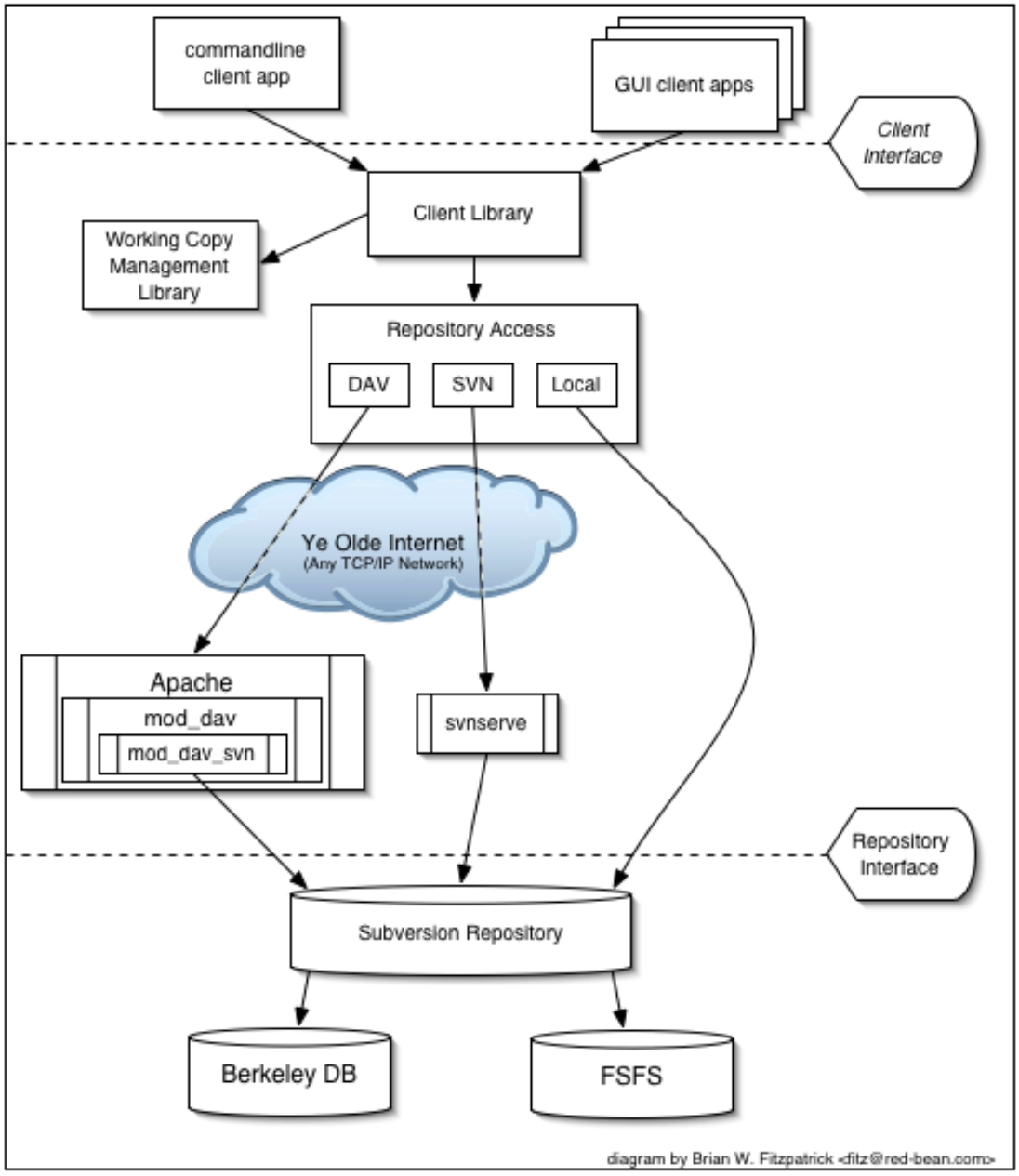
Hackability
: Subversion has no historical baggage; it is implemented as a collection of shared C libraries with well-defined APIs. This makes Subversion extremely maintainable and usable by

other applications and languages.

Subversion's Architecture

Figure 1, “Subversion's Architecture” illustrates a “mile-high” view of Subversion's design.

Figure 1. Subversion's Architecture



On one end is a Subversion repository that holds all of your versioned data. On the other end is your Subversion client program, which manages local reflections of portions of that ver-

sioned data (called “working copies”). Between these extremes are multiple routes through various Repository Access (RA) layers. Some of these routes go across computer networks and through network servers which then access the repository. Others bypass the network altogether and access the repository directly.

Subversion's Components

Subversion, once installed, has a number of different pieces. The following is a quick overview of what you get. Don't be alarmed if the brief descriptions leave you scratching your head—there are *plenty* more pages in this book devoted to alleviating that confusion.

svn
: The command-line client program.

svnversion
: A program for reporting the state (in terms of revisions of the items present) of a working copy.

svnlook
: A tool for directly inspecting a Subversion repository.

svnadmin
: A tool for creating, tweaking or repairing a Subversion repository.

svndumpfilter
: A program for filtering Subversion repository dump streams.

mod_dav_svn
: A plug-in module for the Apache HTTP Server, used to make your repository available to others over a network.

svnserve
: A custom standalone server program, runnable as a daemon process or invokable by SSH; another way to make your repository available to others over a network.

svnsync
: A program for incrementally mirroring one repository to another over a network.

Assuming you have Subversion installed correctly, you should be ready to start. The next two chapters will walk you through the use of **svn**, Subversion's command-line client program.

Chapter 1. Fundamental Concepts

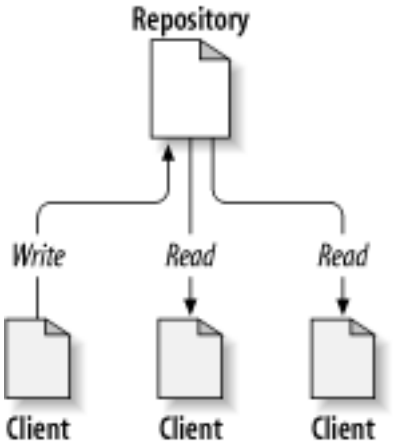
This chapter is a short, casual introduction to Subversion. If you're new to version control, this chapter is definitely for you. We begin with a discussion of general version control concepts, work our way into the specific ideas behind Subversion, and show some simple examples of Subversion in use.

Even though the examples in this chapter show people sharing collections of program source code, keep in mind that Subversion can manage any sort of file collection—it's not limited to helping computer programmers.

The Repository

Subversion is a centralized system for sharing information. At its core is a repository, which is a central store of data. The repository stores information in the form of a *filesystem tree*—a typical hierarchy of files and directories. Any number of *clients* connect to the repository, and then read or write to these files. By writing data, a client makes the information available to others; by reading data, the client receives information from others. Figure 1.1, “A typical client/server system” illustrates this.

Figure 1.1. A typical client/server system



So why is this interesting? So far, this sounds like the definition of a typical file server. And indeed, the repository *is* a kind of file server, but it's not your usual breed. What makes the Subversion repository special is that *it remembers every change* ever written to it: every change to every file, and even changes to the directory tree itself, such as the addition, deletion, and rearrangement of files and directories.

When a client reads data from the repository, it normally sees only the latest version of the filesystem tree. But the client also has the ability to view *previous* states of the filesystem. For example, a client can ask historical questions like, “What did this directory contain last Wednesday?” or “Who was the last person to change this file, and what changes did he make?” These are the sorts of questions that are at the heart of any *version control system*: systems that are designed to track changes to data over time.

Versioning Models

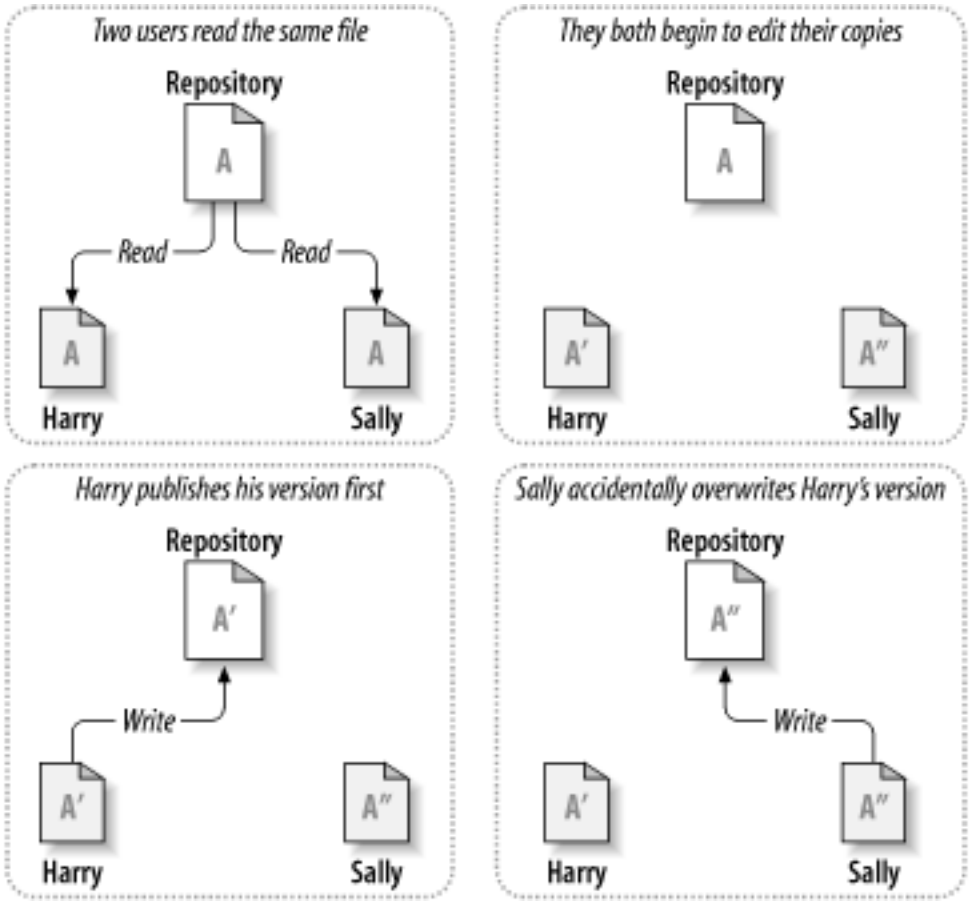
The core mission of a version control system is to enable collaborative editing and sharing of data. But different systems use different strategies to achieve this. It's important to understand these different strategies for a couple of reasons. First, it will help you compare and contrast existing version control systems, in case you encounter other systems similar to Subversion. Beyond that, it will also help you make more effective use of Subversion, since Subversion itself supports a couple of different ways of working.

The Problem of File-Sharing

All version control systems have to solve the same fundamental problem: how will the system allow users to share information, but prevent them from accidentally stepping on each other's feet? It's all too easy for users to accidentally overwrite each other's changes in the repository.

Consider the scenario shown in Figure 1.2, “The problem to avoid”. Suppose we have two coworkers, Harry and Sally. They each decide to edit the same repository file at the same time. If Harry saves his changes to the repository first, then it's possible that (a few moments later) Sally could accidentally overwrite them with her own new version of the file. While Harry's version of the file won't be lost forever (because the system remembers every change), any changes Harry made *won't* be present in Sally's newer version of the file, because she never saw Harry's changes to begin with. Harry's work is still effectively lost—or at least missing from the latest version of the file—and probably by accident. This is definitely a situation we want to avoid!

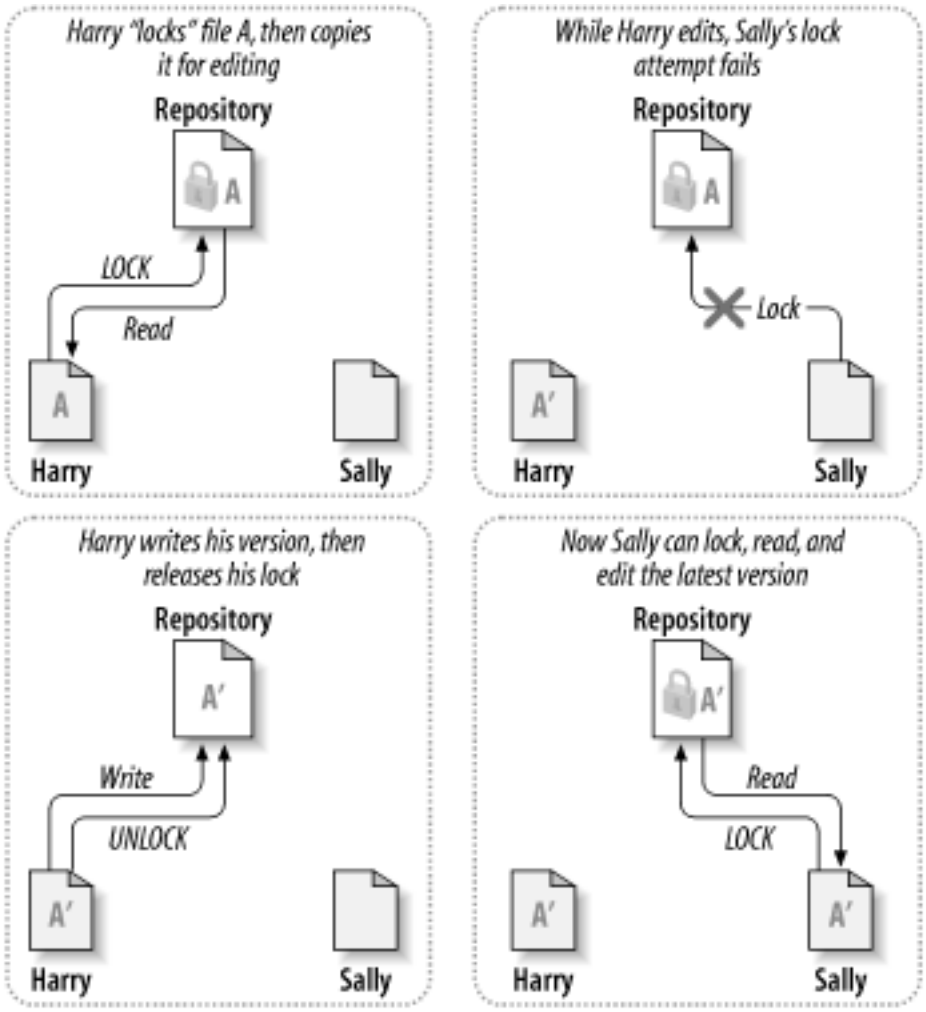
Figure 1.2. The problem to avoid



The Lock-Modify-Unlock Solution

Many version control systems use a *lock-modify-unlock* model to address the problem of many authors clobbering each other's work. In this model, the repository allows only one person to change a file at a time. This exclusivity policy is managed using locks. Harry must “lock” a file before he can begin making changes to it. If Harry has locked a file, then Sally cannot also lock it, and therefore cannot make any changes to that file. All she can do is read the file, and wait for Harry to finish his changes and release his lock. After Harry unlocks the file, Sally can take her turn by locking and editing the file. Figure 1.3, “The lock-modify-unlock solution” demonstrates this simple solution.

Figure 1.3. The lock-modify-unlock solution



The problem with the lock-modify-unlock model is that it's a bit restrictive, and often becomes a roadblock for users:

- *Locking may cause administrative problems.* Sometimes Harry will lock a file and then forget about it. Meanwhile, because Sally is still waiting to edit the file, her hands are tied. And then Harry goes on vacation. Now Sally has to get an administrator to release Harry's lock. The situation ends up causing a lot of unnecessary delay and wasted time.

- *Locking may cause unnecessary serialization.* What if Harry is editing the beginning of a text file, and Sally simply wants to edit the end of the same file? These changes don't overlap at all. They could easily edit the file simultaneously, and no great harm would come, assuming the changes were properly merged together. There's no need for them to take turns in this situation.

- *Locking may create a false sense of security.* Suppose Harry locks and edits file A, while Sally simultaneously locks and edits file B. But what if A and B depend on one another, and the changes made to each are semantically incompatible? Suddenly A and B don't work together anymore. The locking system was powerless to prevent the problem—yet it somehow provided a false sense of security. It's easy for Harry and Sally to imagine that by locking files, each is beginning a safe, insulated task, and thus not bother discussing their incompatible changes early on. Locking often becomes a substitute for real communication.

The Copy-Modify-Merge Solution

Subversion, CVS, and many other version control systems use a *copy-modify-merge* model as an alternative to locking. In this model, each user's client contacts the project repository and creates a personal *working copy*—a local reflection of the repository's files and directories. Users then work simultaneously and independently, modifying their private copies. Finally, the private copies are merged together into a new, final version. The version control system often assists with the merging, but ultimately a human being is responsible for making it happen correctly.

Here's an example. Say that Harry and Sally each create working copies of the same project, copied from the repository. They work concurrently, and make changes to the same file A within their copies. Sally saves her changes to the repository first. When Harry attempts to save his changes later, the repository informs him that his file A is *out-of-date*. In other words, that file A in the repository has somehow changed since he last copied it. So Harry asks his client to *merge* any new changes from the repository into his working copy of file A. Chances are that Sally's changes don't overlap with his own; so once he has both sets of changes integrated, he saves his working copy back to the repository. Figure 1.4, “The copy-modify-merge solution” and Figure 1.5, “The copy-modify-merge solution (continued)” show this process.

Figure 1.4. The copy-modify-merge solution

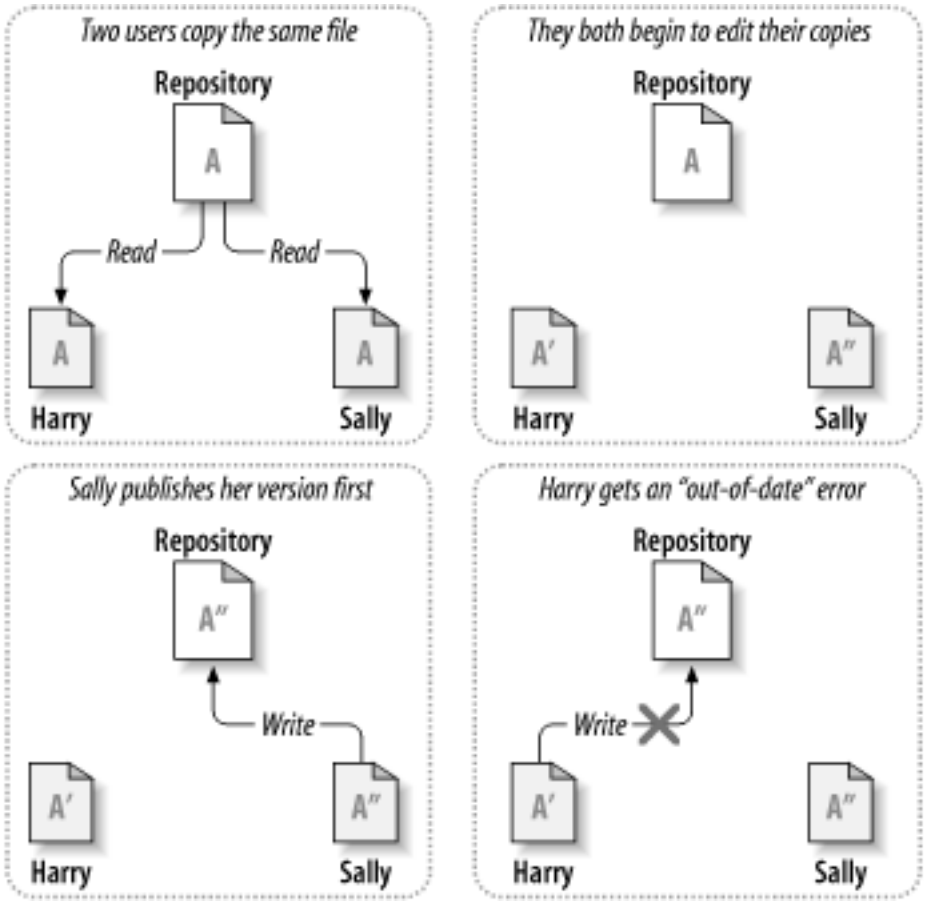
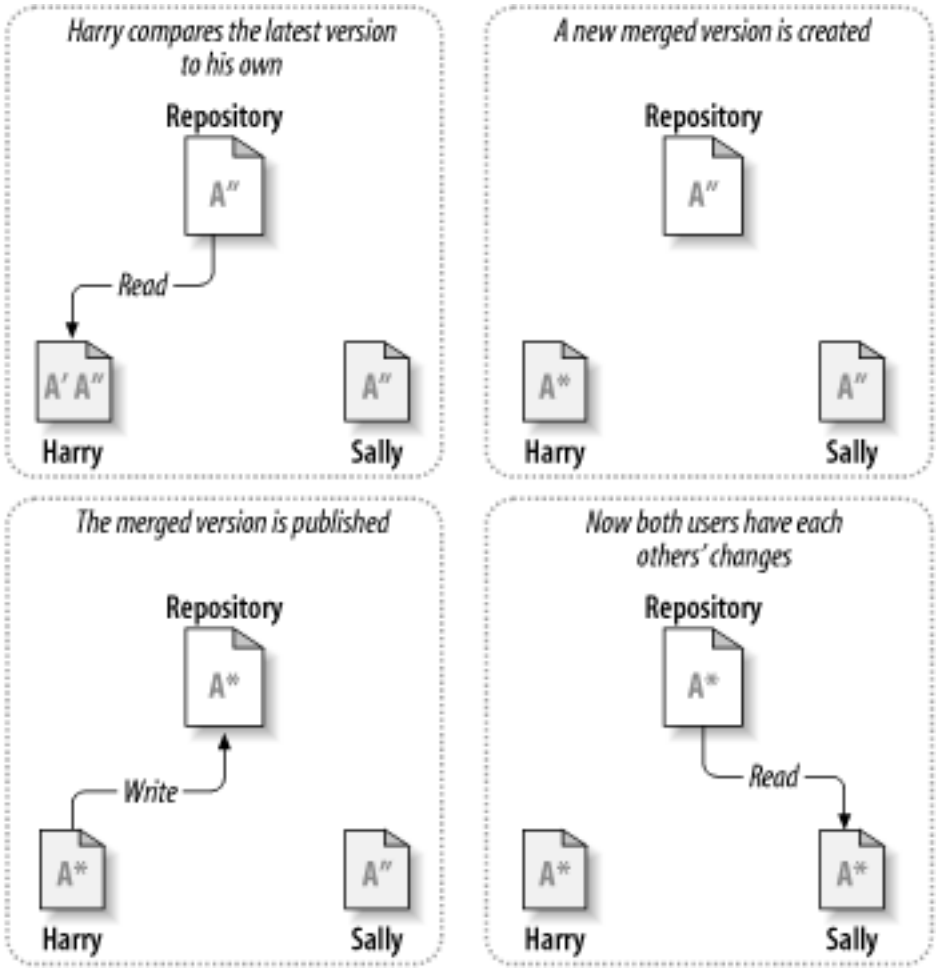


Figure 1.5. The copy-modify-merge solution (continued)



But what if Sally's changes *do* overlap with Harry's changes? What then? This situation is called a *conflict*, and it's usually not much of a problem. When Harry asks his client to merge the latest repository changes into his working copy, his copy of file A is somehow flagged as being in a state of conflict: he'll be able to see both sets of conflicting changes, and manually choose between them. Note that software can't automatically resolve conflicts; only humans are capable of understanding and making the necessary intelligent choices. Once Harry has manually resolved the overlapping changes—perhaps after a discussion with Sally—he can safely save the merged file back to the repository.

The copy-modify-merge model may sound a bit chaotic, but in practice, it runs extremely smoothly. Users can work in parallel, never waiting for one another. When they work on the same files, it turns out that most of their concurrent changes don't overlap at all; conflicts are infrequent. And the amount of time it takes to resolve conflicts is usually far less than the time lost by a locking system.

In the end, it all comes down to one critical factor: user communication. When users communicate poorly, both syntactic and semantic conflicts increase. No system can force users to communicate perfectly, and no system can detect semantic conflicts. So there's no point in being lulled into a false sense of security that a locking system will somehow prevent conflicts; in practice, locking seems to inhibit productivity more than anything else.

When Locking is Necessary

While the lock-modify-unlock model is considered generally harmful to collaboration, there are still times when locking is appropriate.

The copy-modify-merge model is based on the assumption that files are contextually mergeable: that is, that the majority of the files in the repository are line-based text files (such as program source code). But for files with binary formats, such as artwork or sound, it's often impossible to merge conflicting changes. In these situations, it really is necessary to users to take strict turns when changing the file. Without serialized access, somebody ends up wasting time on changes that are ultimately discarded.

While Subversion is still primarily a copy-modify-merge system, it still recognizes the need to lock an occasional file and thus provide mechanisms for this. This feature is discussed later in this book, in the section called “Locking”.

Subversion in Action

It's time to move from the abstract to the concrete. In this section, we'll show real examples of Subversion being used.

Subversion Repository URLs

Throughout this book, Subversion uses URLs to identify versioned files and directories in Subversion repositories. For the most part, these URLs use the standard syntax, allowing for server names and port numbers to be specified as part of the URL:

```
$ svn checkout http://svn.example.com:9834/repos
…
```

But there are some nuances in Subversion's handling of URLs that are notable. For example, URLs containing the `file://` access method (used for local repositories) must, in accordance with convention, have either a server name of `localhost` or no server name at all:

```
$ svn checkout file:///path/to/repos
…
$ svn checkout file://localhost/path/to/repos
…
```

Also, users of the `file://` scheme on Windows platforms will need to use an unofficially “standard” syntax for accessing repositories that are on the same machine, but on a different drive than the client's current working drive. Either of the two following URL path syntaxes will work where X is the drive on which the repository resides:

```
C:\> svn checkout file:///X:/path/to/repos
…
C:\> svn checkout "file:///X|/path/to/repos"
…
```

In the second syntax, you need to quote the URL so that the vertical bar character is not interpreted as a pipe. Also, note that a URL uses forward slashes even though the native (non-URL) form of a path on Windows uses backslashes.



Subversion's `file://` URLs cannot be used in a regular web browser the way typical `file://` URLs can. When you attempt to view a `file://` URL in a regular web browser, it reads and displays the contents of the file at that location by examining the filesystem directly. However, Subversion's resources exist in a virtual filesystem (see the section called “Repository Layer”), and your browser will not understand how to interact with that filesystem.

Finally, it should be noted that the Subversion client will automatically encode URLs as necessary, just like a web browser does. For example, if a URL contains a space or upper-ASCII character:

```
$ svn checkout "http://host/path with space/project/españa"
```

…then Subversion will escape the unsafe characters and behave as if you had typed:

```
$ svn checkout http://host/path%20with%20space/project/espa%C3%B1a
```

If the URL contains spaces, be sure to place it within quote marks, so that your shell treats the whole thing as a single argument to the **svn** program.

Working Copies

You've already read about working copies; now we'll demonstrate how the Subversion client creates and uses them.

A Subversion working copy is an ordinary directory tree on your local system, containing a collection of files. You can edit these files however you wish, and if they're source code files, you can compile your program from them in the usual way. Your working copy is your own private work area: Subversion will never incorporate other people's changes, nor make your own changes available to others, until you explicitly tell it to do so. You can even have multiple working copies of the same project.

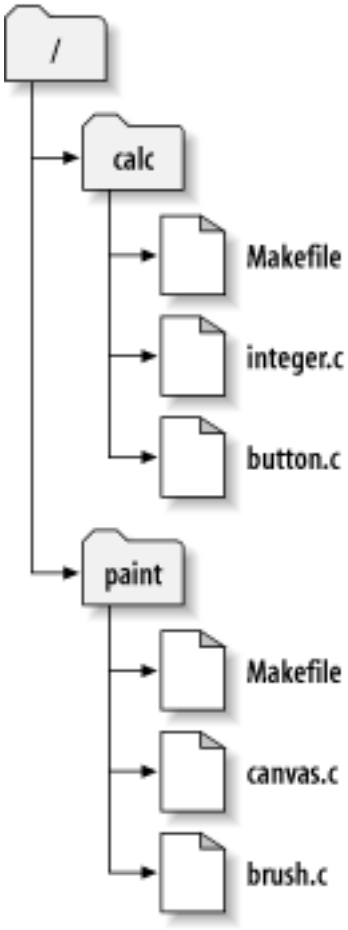
After you've made some changes to the files in your working copy and verified that they work properly, Subversion provides you with commands to “publish” your changes to the other people working with you on your project (by writing to the repository). If other people publish their own changes, Subversion provides you with commands to merge those changes into your working directory (by reading from the repository).

A working copy also contains some extra files, created and maintained by Subversion, to help it carry out these commands. In particular, each directory in your working copy contains a subdirectory named `.svn`, also known as the working copy *administrative directory*. The files in each administrative directory help Subversion recognize which files contain unpublished changes, and which files are out-of-date with respect to others' work.

A typical Subversion repository often holds the files (or source code) for several projects; usually, each project is a subdirectory in the repository's filesystem tree. In this arrangement, a user's working copy will usually correspond to a particular subtree of the repository.

For example, suppose you have a repository that contains two software projects, `paint` and `calc`. Each project lives in its own top-level subdirectory, as shown in Figure 1.6, “The repository's filesystem”.

Figure 1.6. The repository's filesystem



To get a working copy, you must *check out* some subtree of the repository. (The term “check out” may sound like it has something to do with locking or reserving resources, but it doesn't; it simply creates a private copy of the project for you.) For example, if you check out `/calc`, you will get a working copy like this:

```
$ svn checkout http://svn.example.com/repos/calc
A    calc/Makefile
A    calc/integer.c
A    calc/button.c
Checked out revision 56.

$ ls -A calc
Makefile  integer.c  button.c  .svn/
```

The list of letter A's in the left margin indicates that Subversion is adding a number of items to your working copy. You now have a personal copy of the repository's `/calc` directory, with one additional entry—`.svn`—which holds the extra information needed by Subversion, as mentioned earlier.

Suppose you make changes to `button.c`. Since the `.svn` directory remembers the file's ori-

ginal modification date and contents, Subversion can tell that you've changed the file. However, Subversion does not make your changes public until you explicitly tell it to. The act of publishing your changes is more commonly known as *committing* (or *checking in*) changes to the repository.

To publish your changes to others, you can use Subversion's **commit** command.

```
$ svn commit button.c -m "Fixed a typo in button.c."
Sending        button.c
Transmitting file data .
Committed revision 57.
```

Now your changes to `button.c` have been committed to the repository, with a note describing your change (namely, that you fixed a typo). If another user checks out a working copy of `/calc`, they will see your changes in the latest version of the file.

Suppose you have a collaborator, Sally, who checked out a working copy of `/calc` at the same time you did. When you commit your change to `button.c`, Sally's working copy is left unchanged; Subversion only modifies working copies at the user's request.

To bring her project up to date, Sally can ask Subversion to *update* her working copy, by using the Subversion **update** command. This will incorporate your changes into her working copy, as well as any others that have been committed since she checked it out.

```
$ pwd
/home/sally/calc

$ ls -A
.svn/ Makefile integer.c button.c

$ svn update
U    button.c
Updated to revision 57.
```

The output from the **svn update** command indicates that Subversion updated the contents of `button.c`. Note that Sally didn't need to specify which files to update; Subversion uses the information in the `.svn` directory, and further information in the repository, to decide which files need to be brought up to date.

Repository URLs

Subversion repositories can be accessed through many different methods—on local disk, or through various network protocols, depending on how your administrator has set things up for you. A repository location, however, is always a URL. Table 1.1, “Repository Access URLs” describes how different URL schemes map to the available access methods.

Table 1.1. Repository Access URLs

Schema	Access Method
`file:///`	direct repository access (on local disk)
`http://`	access via WebDAV protocol to Subversion-aware Apache server

Schema	Access Method
`https://`	same as `http://`, but with SSL encryption.
`svn://`	access via custom protocol to an `svnserve` server
`svn+ssh://`	same as `svn://`, but through an SSH tunnel.

For more information on how Subversion parses URLs, see the section called “Subversion Repository URLs”. For more information on the different types of network servers available for Subversion, see Chapter 6, *Server Configuration*.

Revisions

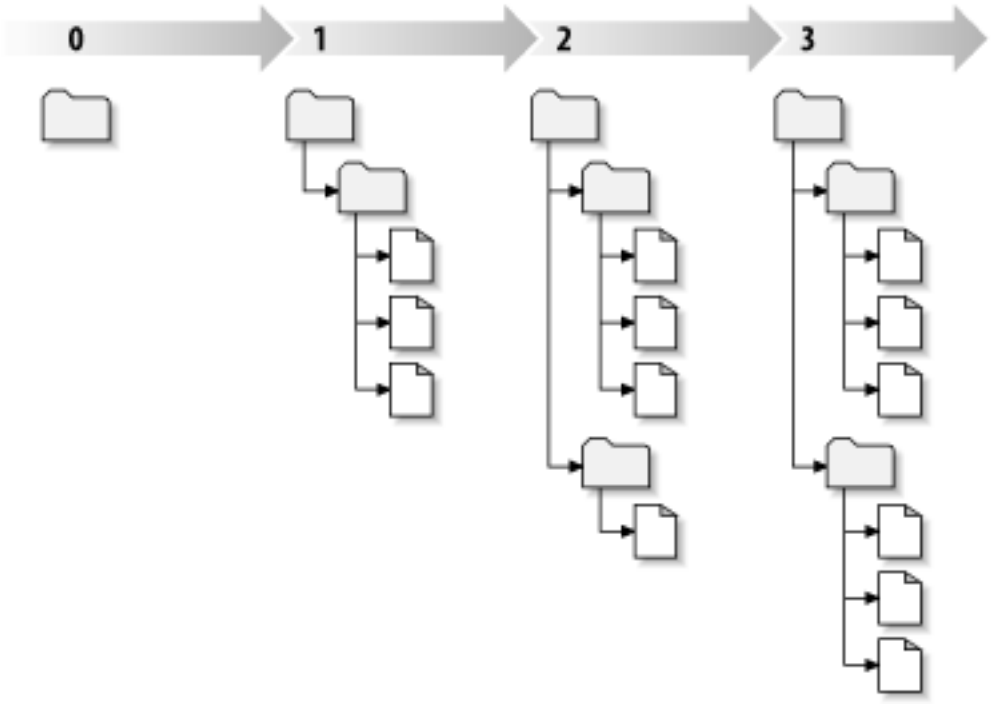
An **svn commit** operation publishes changes to any number of files and directories as a single atomic transaction. In your working copy, you can change files' contents; create, delete, rename and copy files and directories; then commit a complete set of changes as an atomic transaction.

By “atomic transaction”, we mean simply this: either all of the changes happen in the repository, or none of them happen. Subversion tries to retain this atomicity in the face of program crashes, system crashes, network problems, and other users' actions.

Each time the repository accepts a commit, this creates a new state of the filesystem tree, called a *revision*. Each revision is assigned a unique natural number, one greater than the number of the previous revision. The initial revision of a freshly created repository is numbered zero, and consists of nothing but an empty root directory.

Figure 1.7, “The repository” illustrates a nice way to visualize the repository. Imagine an array of revision numbers, starting at 0, stretching from left to right. Each revision number has a filesystem tree hanging below it, and each tree is a “snapshot” of the way the repository looked after a commit.

Figure 1.7. The repository



Global Revision Numbers

Unlike most version control systems, Subversion's revision numbers apply to *entire trees*, not individual files. Each revision number selects an entire tree, a particular state of the repository after some committed change. Another way to think about it is that revision N represents the state of the repository filesystem after the Nth commit. When Subversion users talk about “revision 5 of `foo.c`”, they really mean “`foo.c` as it appears in revision 5.” Notice that in general, revisions N and M of a file do *not* necessarily differ! Many other version control systems use per-file revision numbers, so this concept may seem unusual at first. (Former CVS users might want to see Appendix B, *Subversion for CVS Users* for more details.)

It's important to note that working copies do not always correspond to any single revision in the repository; they may contain files from several different revisions. For example, suppose you check out a working copy from a repository whose most recent revision is 4:

```
calc/Makefile:4
     integer.c:4
     button.c:4
```

At the moment, this working directory corresponds exactly to revision 4 in the repository. However, suppose you make a change to `button.c`, and commit that change. Assuming no other commits have taken place, your commit will create revision 5 of the repository, and your working copy will now look like this:

```
calc/Makefile:4
     integer.c:4
     button.c:5
```

Suppose that, at this point, Sally commits a change to `integer.c`, creating revision 6. If you use **svn update** to bring your working copy up to date, then it will look like this:

```
calc/Makefile:6
     integer.c:6
     button.c:6
```

Sally's change to `integer.c` will appear in your working copy, and your change will still be present in `button.c`. In this example, the text of `Makefile` is identical in revisions 4, 5, and 6, but Subversion will mark your working copy of `Makefile` with revision 6 to indicate that it is still current. So, after you do a clean update at the top of your working copy, it will generally correspond to exactly one revision in the repository.

How Working Copies Track the Repository

For each file in a working directory, Subversion records two essential pieces of information in the `.svn/` administrative area:

- what revision your working file is based on (this is called the file's *working revision*), and
- a timestamp recording when the local copy was last updated by the repository.

Given this information, by talking to the repository, Subversion can tell which of the following four states a working file is in:

Unchanged, and current
: The file is unchanged in the working directory, and no changes to that file have been committed to the repository since its working revision. An **svn commit** of the file will do nothing, and an **svn update** of the file will do nothing.

Locally changed, and current
: The file has been changed in the working directory, and no changes to that file have been committed to the repository since you last updated. There are local changes that have not been committed to the repository, thus an **svn commit** of the file will succeed in publishing your changes, and an **svn update** of the file will do nothing.

Unchanged, and out-of-date
: The file has not been changed in the working directory, but it has been changed in the repository. The file should eventually be updated, to make it current with the latest public revision. An **svn commit** of the file will do nothing, and an **svn update** of the file will fold the latest changes into your working copy.

Locally changed, and out-of-date
: The file has been changed both in the working directory, and in the repository. An **svn commit** of the file will fail with an “out-of-date” error. The file should be updated first; an **svn update** command will attempt to merge the public changes with the local changes. If Subversion can't complete the merge in a plausible way automatically, it leaves it to the user to resolve the conflict.

This may sound like a lot to keep track of, but the **svn status** command will show you the state

of any item in your working copy. For more information on that command, see the section called “See an overview of your changes”.

Mixed Revision Working Copies

As a general principle, Subversion tries to be as flexible as possible. One special kind of flexibility is the ability to have a working copy containing files and directories with a mix of different working revision numbers. Unfortunately, this flexibility tends to confuse a number of new users. If the earlier example showing mixed revisions perplexed you, here's a primer on both why the feature exists and how to make use of it.

Updates and Commits are Separate

One of the fundamental rules of Subversion is that a “push” action does not cause a “pull”, nor the other way around. Just because you're ready to submit new changes to the repository doesn't mean you're ready to receive changes from other people. And if you have new changes still in progress, then **svn update** should gracefully merge repository changes into your own, rather than forcing you to publish them.

The main side-effect of this rule is that it means a working copy has to do extra bookkeeping to track mixed revisions, and be tolerant of the mixture as well. It's made more complicated by the fact that directories themselves are versioned.

For example, suppose you have a working copy entirely at revision 10. You edit the file `foo.html` and then perform an **svn commit**, which creates revision 15 in the repository. After the commit succeeds, many new users would expect the working copy to be entirely at revision 15, but that's not the case! Any number of changes might have happened in the repository between revisions 10 and 15. The client knows nothing of those changes in the repository, since you haven't yet run **svn update**, and **svn commit** doesn't pull down new changes. If, on the other hand, **svn commit** *were* to automatically download the newest changes, then it would be possible to set the entire working copy to revision 15—but then we'd be breaking the fundamental rule of “push” and “pull” remaining separate actions. Therefore the only safe thing the Subversion client can do is mark the one file—`foo.html`—as being at revision 15. The rest of the working copy remains at revision 10. Only by running **svn update** can the latest changes be downloaded, and the whole working copy be marked as revision 15.

Mixed revisions are normal

The fact is, *every time* you run **svn commit**, your working copy ends up with some mixture of revisions. The things you just committed are marked as having larger working revisions than everything else. After several commits (with no updates in-between) your working copy will contain a whole mixture of revisions. Even if you're the only person using the repository, you will still see this phenomenon. To examine your mixture of working revisions, use the **svn status --verbose** command (see the section called “See an overview of your changes” for more information.)

Often, new users are completely unaware that their working copy contains mixed revisions. This can be confusing, because many client commands are sensitive to the working revision of the item they're examining. For example, the **svn log** command is used to display the history of changes to a file or directory (see the section called “Generating a list of historical changes”). When the user invokes this command on a working copy object, they expect to see the entire history of the object. But if the object's working revision is quite old (often because **svn update** hasn't been run in a long time), then the history of the *older* version of the object is shown.

Mixed revisions are useful

If your project is sufficiently complex, you'll discover that it's sometimes nice to forcibly *back-date* (or, update to a revision older than the one you already have) portions of your working copy to an earlier revision; you'll learn how to do that in Chapter 2, *Basic Usage*. Perhaps you'd like to test an earlier version of a sub-module contained in a subdirectory, or perhaps you'd like to figure out when a bug first came into existence in a specific file. This is the “time machine” aspect of a version control system—the feature which allows you to move any portion of your working copy forward and backward in history.

Mixed revisions have limitations

However you make use of mixed revisions in your working copy, there are limitations to this flexibility.

First, you cannot commit the deletion of a file or directory which isn't fully up-to-date. If a newer version of the item exists in the repository, your attempt to delete will be rejected, to prevent you from accidentally destroying changes you've not yet seen.

Second, you cannot commit a metadata change to a directory unless it's fully up-to-date. You'll learn about attaching “properties” to items in Chapter 3, *Advanced Topics*. A directory's working revision defines a specific set of entries and properties, and thus committing a property change to an out-of-date directory may destroy properties you've not yet seen.

Summary

We've covered a number of fundamental Subversion concepts in this chapter:

- We've introduced the notions of the central repository, the client working copy, and the array of repository revision trees.
- We've seen some simple examples of how two collaborators can use Subversion to publish and receive changes from one another, using the “copy-modify-merge” model.
- We've talked a bit about the way Subversion tracks and manages information in a working copy.

At this point, you should have a good idea of how Subversion works in the most general sense. Armed with this knowledge, you should now be ready to move into the next chapter, which is a detailed tour of Subversion's commands and features.

Chapter 2. Basic Usage

Now we will go into the details of using Subversion. By the time you reach the end of this chapter, you will be able to perform all the tasks you need to use Subversion in a normal day's work. You'll start with getting your files into Subversion, followed by an initial checkout of your code. We'll then walk you through making changes and examining those changes. You'll also see how to bring changes made by others into your working copy, examine them, and work through any conflicts that might arise.

Note that this chapter is not meant to be an exhaustive list of all Subversion's commands—rather, it's a conversational introduction to the most common Subversion tasks you'll encounter. This chapter assumes that you've read and understood Chapter 1, *Fundamental Concepts* and are familiar with the general model of Subversion. For a complete reference of all commands, see Chapter 9, *Subversion Complete Reference*.

Help!

Before reading on, here is the most important command you'll ever need when using Subversion: **svn help**. The Subversion command-line client is self-documenting—at any time, a quick **svn help *SUBCOMMAND*** will describe the syntax, options, and behavior of the subcommand.

```
$ svn help import
import: Commit an unversioned file or tree into the repository.
usage: import [PATH] URL

  Recursively commit a copy of PATH to URL.
  If PATH is omitted '.' is assumed.
  Parent directories are created as necessary in the repository.
  If PATH is a directory, the contents of the directory are added
  directly under URL.

Valid options:
  -q [--quiet]             : print as little as possible
  -N [--non-recursive]     : operate on single directory only
…
```

Getting Data into your Repository

There are two ways to get new files into your Subversion repository: **svn import** and **svn add**. We'll discuss **svn import** here and **svn add** later in this chapter when we review a typical day with Subversion.

svn import

The **svn import** command is a quick way to copy an unversioned tree of files into a repository, creating intermediate directories as necessary. **svn import** doesn't require a working copy, and your files are immediately committed to the repository. This is typically used when you have an existing tree of files that you want to begin tracking in your Subversion repository. For example:

```
$ svnadmin create /usr/local/svn/newrepos
$ svn import mytree file:///usr/local/svn/newrepos/some/project \
```

```
             -m "Initial import"
Adding         mytree/foo.c
Adding         mytree/bar.c
Adding         mytree/subdir
Adding         mytree/subdir/quux.h

Committed revision 1.
```

The previous example copied the contents of directory `mytree` under the directory `some/project` in the repository:

```
$ svn list file:///usr/local/svn/newrepos/some/project
bar.c
foo.c
subdir/
```

Note that after the import is finished, the original tree is *not* converted into a working copy. To start working, you still need to **svn checkout** a fresh working copy of the tree.

Recommended repository layout

While Subversion's flexibility allows you to layout your repository in any way that you choose, we recommend that you create a `trunk` directory to hold the “main line” of development, a `branches` directory to contain branch copies, and a `tags` directory to contain tag copies, for example:

```
$ svn list file:///usr/local/svn/repos
/trunk
/branches
/tags
```

You'll learn more about tags and branches in Chapter 4, *Branching and Merging*. For details and how to set up multiple projects, see the section called “Repository Layout” and the section called “Planning Your Repository Organization” to read more about “project roots”.

Initial Checkout

Most of the time, you will start using a Subversion repository by doing a *checkout* of your project. Checking out a repository creates a “working copy” of it on your local machine. This copy contains the `HEAD` (latest revision) of the Subversion repository that you specify on the command line:

```
$ svn checkout http://svn.collab.net/repos/svn/trunk
A    trunk/Makefile.in
A    trunk/ac-helpers
A    trunk/ac-helpers/install.sh
A    trunk/ac-helpers/install-sh
A    trunk/build.conf
…
Checked out revision 8810.
```

What's in a Name?

Subversion tries hard not to limit the type of data you can place under version control. The contents of files and property values are stored and transmitted as binary data, and the section called “File Content Type” tells you how to give Subversion a hint that “textual” operations don't make sense for a particular file. There are a few places, however, where Subversion places restrictions on information it stores.

Subversion internally handles certain bits of data—for example, property names, path names, and log messages—as UTF-8 encoded Unicode. This is not to say that all your interactions with Subversion must involve UTF-8, though. As a general rule, Subversion clients will gracefully and transparently handle conversions between UTF-8 and the encoding system in use on your computer, if such a conversion can meaningfully be done (which is the case for most common encodings in use today).

In addition, path names are used as XML attribute values in WebDAV exchanges, as well in as some of Subversion's housekeeping files. This means that path names can only contain legal XML (1.0) characters. Subversion also prohibits TAB, CR, and LF characters in path names to prevent paths from being broken up in diffs, or in the output of commands like svn log or svn status.

While it may seem like a lot to remember, in practice these limitations are rarely a problem. As long as your locale settings are compatible with UTF-8, and you don't use control characters in path names, you should have no trouble communicating with Subversion. The command-line client adds an extra bit of help—it will automatically escape illegal path characters as needed in URLs you type to create “legally correct” versions for internal use.

Although the above example checks out the trunk directory, you can just as easily check out any deep subdirectory of a repository by specifying the subdirectory in the checkout URL:

```
$ svn checkout \
     http://svn.collab.net/repos/svn/trunk/subversion/tests/cmdline/
A    cmdline/revert_tests.py
A    cmdline/diff_tests.py
A    cmdline/autoprop_tests.py
A    cmdline/xmltests
A    cmdline/xmltests/svn-test.sh
…
Checked out revision 8810.
```

Since Subversion uses a “copy-modify-merge” model instead of “lock-modify-unlock” (see the section called “Versioning Models”), you can start right in making changes to the files and directories in your working copy. Your working copy is just like any other collection of files and directories on your system. You can edit and change them, move them around, you can even delete the entire working copy and forget about it.



While your working copy is “just like any other collection of files and directories on your system”, you can edit files at will, but you must tell Subversion about *everything else* that you do. For example, if you want to copy or move an item in a working copy, you should use **svn copy** or **svn move** instead of the copy and move commands provided by your operating system. We'll talk more about them later in this chapter.

Unless you're ready to commit the addition of a new file or directory, or changes to existing ones, there's no need to further notify the Subversion server that you've done anything.

> **What's with the `.svn` directory?**
>
> Every directory in a working copy contains an administrative area, a subdirectory named `.svn`. Usually, directory listing commands won't show this subdirectory, but it is nevertheless an important directory. Whatever you do, don't delete or change anything in the administrative area! Subversion depends on it to manage your working copy.
>
> If you accidentally remove the `.svn` subdirectory, the easiest way to fix the problem is to remove the entire containing directory (a normal system deletion, not **svn delete**), then run **svn update** from a parent directory. The Subversion client will re-download the directory you've deleted, with a new `.svn` area as well.

While you can certainly check out a working copy with the URL of the repository as the only argument, you can also specify a directory after your repository URL. This places your working copy in the new directory that you name. For example:

```
$  svn checkout http://svn.collab.net/repos/svn/trunk subv
A    subv/Makefile.in
A    subv/ac-helpers
A    subv/ac-helpers/install.sh
A    subv/ac-helpers/install-sh
A    subv/build.conf
…
Checked out revision 8810.
```

That will place your working copy in a directory named `subv` instead of a directory named `trunk` as we did previously. The directory `subv` will be created if it doesn't already exist.

Disabling Password Caching

When you perform a Subversion operation that requires you to authenticate, by default Subversion caches your authentication credentials on disk. This is done for convenience, so that you don't have to continually re-enter your password for future operations. If you're concerned about caching your Subversion passwords,[1] you can disable caching either permanently or on a case-by-case basis.

To disable password caching for a particular one-time command, pass the `--no-auth-cache` option on the commandline. To permanently disable caching, you can add the line `store-passwords = no` to your local machine's Subversion configuration file. See the section called “Client Credentials Caching” for details.

Authenticating as a Different User

Since Subversion caches auth credentials by default (both username and password), it conveniently remembers who you were acting as the last time you modified you working copy. But sometimes that's not helpful—particularly if you're working in a shared working copy, like a system configuration directory or a webserver document root. In this case, just pass the `-`

[1] Of course, you're not terribly worried—first because you know that you can't *really* delete anything from Subversion and, secondly, because your Subversion password isn't the same as any of the other three million passwords you have, right? Right?

`-username` option on the commandline and Subversion will attempt to authenticate as that user, prompting you for a password if necessary.

Basic Work Cycle

Subversion has numerous features, options, bells and whistles, but on a day-to-day basis, odds are that you will only use a few of them. In this section we'll run through the most common things that you might find yourself doing with Subversion in the course of a day's work.

The typical work cycle looks like this:

- Update your working copy
  - **svn update**
- Make changes
  - **svn add**
  - **svn delete**
  - **svn copy**
  - **svn move**
- Examine your changes
  - **svn status**
  - **svn diff**
- Possibly undo some changes
  - **svn revert**
- Resolve Conflicts (Merge Others' Changes)
  - **svn update**
  - **svn resolved**
- Commit your changes
  - **svn commit**

Update Your Working Copy

When working on a project with a team, you'll want to update your working copy to receive any changes made since your last update by other developers on the project. Use **svn update** to bring your working copy into sync with the latest revision in the repository.

```
$ svn update
U  foo.c
U  bar.c
Updated to revision 2.
```

In this case, someone else checked in modifications to both `foo.c` and `bar.c` since the last time you updated, and Subversion has updated your working copy to include those changes.

When the server sends changes to your working copy via **svn update**, a letter code is displayed next to each item to let you know what actions Subversion performed to bring your working copy up-to-date. To find out what these letters mean, see svn update.

Make Changes to Your Working Copy

Now you can get to work and make changes in your working copy. It's usually most convenient to decide on a discrete change (or set of changes) to make, such as writing a new feature, fixing a bug, etc. The Subversion commands that you will use here are **svn add**, **svn delete**, **svn copy**, **svn move**, and **svn mkdir**. However, if you are merely editing files that are already in Subversion, you may not need to use any of these commands until you commit.

There are two kinds of changes you can make to your working copy: file changes and tree changes. You don't need to tell Subversion that you intend to change a file; just make your changes using your text editor, word processor, graphics program, or whatever tool you would normally use. Subversion automatically detects which files have been changed, and in addition handles binary files just as easily as it handles text files—and just as efficiently too. For tree changes, you can ask Subversion to “mark” files and directories for scheduled removal, addition, copying, or moving. These changes may take place immediately in your working copy, but no additions or removals will happen in the repository until you commit them.

Here is an overview of the five Subversion subcommands that you'll use most often to make tree changes.

Versioning symbolic links

On non-Windows platforms, Subversion is able to version files of the special type *symbolic link* (or, “symlink”). A symlink is a file which acts as a sort of transparent reference to some other object in the filesystem, allowing programs to read and write to those objects indirectly by way of performing operations on the symlink itself.

When a symlink is committed into a Subversion repository, Subversion remembers that the file was in fact a symlink, as well as the object to which the symlink “points”. When that symlink is checked out to another working copy on a non-Windows system, Subversion reconstructs a real filesystem-level symbolic link from the versioned symlink. But that doesn't in any way limit the usability of working copies on systems such as Windows which do not support symlinks. On such systems, Subversion simply creates a regular text file whose contents are the path to which to the original symlink pointed. While that file can't be used as a symlink on a Windows system, it also won't prevent Windows users from performing their other Subversion-related activities.

svn add foo
: Schedule file, directory, or symbolic link `foo` to be added to the repository. When you next commit, `foo` will become a child of its parent directory. Note that if `foo` is a directory, everything underneath `foo` will be scheduled for addition. If you only want to add `foo` itself, pass the `--non-recursive (-N)` option.

svn delete foo
: Schedule file, directory, or symbolic link `foo` to be deleted from the repository. If `foo` is a file or link, it is immediately deleted from your working copy. If `foo` is a directory, it is not deleted, but Subversion schedules it for deletion. When you commit your changes, `foo` will

be entirely removed from your working copy and the repository. [2]

svn copy foo bar
Create a new item `bar` as a duplicate of `foo` and automatically schedule `bar` for addition. When `bar` is added to the repository on the next commit, its copy history is recorded (as having originally come from `foo`). **svn copy** does not create intermediate directories.

svn move foo bar
This command is exactly the same as running **svn copy foo bar; svn delete foo**. That is, `bar` is scheduled for addition as a copy of `foo`, and `foo` is scheduled for removal. **svn move** does not create intermediate directories.

svn mkdir blort
This command is exactly the same as running **mkdir blort; svn add blort**. That is, a new directory named `blort` is created and scheduled for addition.

Changing the Repository Without a Working Copy

There *are* some use cases that immediately commit tree changes to the repository. This only happens when a subcommand is operating directly on a URL, rather than on a working-copy path. In particular, specific uses of **svn mkdir**, **svn copy**, **svn move**, and **svn delete** can work with URLs (And don't forget that **svn import** always makes changes to a URL).

URL operations behave in this manner because commands that operate on a working copy can use the working copy as a sort of “staging area” to set up your changes before committing them to the repository. Commands that operate on URLs don't have this luxury, so when you operate directly on a URL, any of the above actions represent an immediate commit.

Examine Your Changes

Once you've finished making changes, you need to commit them to the repository, but before you do so, it's usually a good idea to take a look at exactly what you've changed. By examining your changes before you commit, you can make a more accurate log message. You may also discover that you've inadvertently changed a file, and this gives you a chance to revert those changes before committing. Additionally, this is a good opportunity to review and scrutinize changes before publishing them. You can see an overview of the changes you've made by using **svn status**, and dig into the details of those changes by using **svn diff**.

Look Ma! No Network!

The commands **svn status**, **svn diff**, and **svn revert** can be used without any network access even if your repository *is* across the network. This makes it easy to manage your changes-in-progress when you are somewhere without a network connection, such as travelling on an airplane, riding a commuter train or hacking on the beach.[3]

Subversion does this by keeping private caches of pristine versions of each versioned file inside of the `.svn` administrative areas. This allows Subversion to report—and re-

[2] Of course, nothing is ever totally deleted from the repository—just from the `HEAD` of the repository. You can get back anything you delete by checking out (or updating your working copy to) a revision earlier than the one in which you deleted it. Also see the section called “Resurrecting Deleted Items”.
[3] And also that you don't have a WAN card. Thought you got us, huh?

vert—local modifications to those files *without network access*. This cache (called the "text-base") also allows Subversion to send the user's local modifications during a commit to the server as a compressed *delta* (or "difference") against the pristine version. Having this cache is a tremendous benefit—even if you have a fast net connection, it's much faster to send only a file's changes rather than the whole file to the server.

Subversion has been optimized to help you with this task, and is able to do many things without communicating with the repository. In particular, your working copy contains a hidden cached "pristine" copy of each version controlled file within the `.svn` area. Because of this, Subversion can quickly show you how your working files have changed, or even allow you to undo your changes without contacting the repository.

See an overview of your changes

To get an overview of your changes, you'll use the **svn status** command. You'll probably use **svn status** more than any other Subversion command.

CVS Users: Hold That Update!

You're probably used to using **cvs update** to see what changes you've made to your working copy. **svn status** will give you all the information you need regarding what has changed in your working copy—without accessing the repository or potentially incorporating new changes published by other users.

In Subversion, **update** does just that—it updates your working copy with any changes committed to the repository since the last time you've updated your working copy. You may have to break the habit of using the **update** command to see what local modifications you've made.

If you run **svn status** at the top of your working copy with no arguments, it will detect all file and tree changes you've made. Below are a few examples of the most common status codes that **svn status** can return. (Note that the text following `#` is not actually printed by **svn status**.)

```
A       stuff/loot/bloo.h   # file is scheduled for addition
C       stuff/loot/lump.c   # file has textual conflicts from an update
D       stuff/fish.c        # file is scheduled for deletion
M       bar.c               # the content in bar.c has local modifications
```

In this output format **svn status** prints six columns of characters, followed by several whitespace characters, followed by a file or directory name. The first column tells the status of a file or directory and/or its contents. The codes we listed are:

`A item`
: The file, directory, or symbolic link `item` has been scheduled for addition into the repository.

`C item`
: The file `item` is in a state of conflict. That is, changes received from the server during an update overlap with local changes that you have in your working copy. You must resolve this conflict before committing your changes to the repository.

`D item`
The file, directory, or symbolic link `item` has been scheduled for deletion from the repository.

`M item`
The contents of the file `item` have been modified.

If you pass a specific path to **svn status**, you get information about that item alone:

```
$ svn status stuff/fish.c
D      stuff/fish.c
```

svn status also has a `--verbose (-v)` option, which will show you the status of *every* item in your working copy, even if it has not been changed:

```
$ svn status -v
M               44        23    sally     README
                44        30    sally     INSTALL
M               44        20    harry     bar.c
                44        18    ira       stuff
                44        35    harry     stuff/trout.c
D               44        19    ira       stuff/fish.c
                44        21    sally     stuff/things
A                0         ?     ?        stuff/things/bloo.h
                44        36    harry     stuff/things/gloo.c
```

This is the “long form” output of **svn status**. The letters in the first column mean the same as before, but the second column shows the working-revision of the item. The third and fourth columns show the revision in which the item last changed, and who changed it.

None of the prior invocations to **svn status** contact the repository—instead, they compare the metadata in the `.svn` directory with the working copy. Finally, there is the `--show-updates (-u)` option, which contacts the repository and adds information about things that are out-of-date:

```
$ svn status -u -v
M      *        44        23    sally     README
M               44        20    harry     bar.c
       *        44        35    harry     stuff/trout.c
D               44        19    ira       stuff/fish.c
A                0         ?     ?        stuff/things/bloo.h
Status against revision:   46
```

Notice the two asterisks: if you were to run **svn update** at this point, you would receive changes to `README` and `trout.c`. This tells you some very useful information—you'll need to update and get the server changes on `README` before you commit, or the repository will reject your commit for being out-of-date. (More on this subject later.)

svn status can display much more information about the files and directories in your working copy than we've shown here—for an exhaustive description of svn status and its output, see svn status.

Examine the details of your local modifications

Another way to examine your changes is with the **svn diff** command. You can find out *exactly* how you've modified things by running **svn diff** with no arguments, which prints out file changes in *unified diff format*:

```
$ svn diff
Index: bar.c
===================================================================
--- bar.c (revision 3)
+++ bar.c (working copy)
@@ -1,7 +1,12 @@
+#include <sys/types.h>
+#include <sys/stat.h>
+#include <unistd.h>
+
+#include <stdio.h>

 int main(void) {
-  printf("Sixty-four slices of American Cheese...\n");
+  printf("Sixty-five slices of American Cheese...\n");
 return 0;
 }

Index: README
===================================================================
--- README (revision 3)
+++ README (working copy)
@@ -193,3 +193,4 @@
+Note to self:  pick up laundry.

Index: stuff/fish.c
===================================================================
--- stuff/fish.c (revision 1)
+++ stuff/fish.c (working copy)
-Welcome to the file known as 'fish'.
-Information on fish will be here soon.

Index: stuff/things/bloo.h
===================================================================
--- stuff/things/bloo.h (revision 8)
+++ stuff/things/bloo.h (working copy)
+Here is a new file to describe
+things about bloo.
```

The **svn diff** command produces this output by comparing your working files against the cached “pristine” copies within the `.svn` area. Files scheduled for addition are displayed as all added-text, and files scheduled for deletion are displayed as all deleted text.

Output is displayed in unified diff format. That is, removed lines are prefaced with `-` and added lines are prefaced with `+`. **svn diff** also prints filename and offset information useful to the **patch** program, so you can generate “patches” by redirecting the diff output to a file:

```
$ svn diff > patchfile
```

You could, for example, email the patch file to another developer for review or testing prior to commit.

Subversion uses its internal diff engine, which produces unified diff format, by default. If you want diff output in a different format, specify an external diff program using `--diff-cmd` and

pass any flags you'd like to it using the `--extensions (-x)` option. For example, to see local differences in file `foo.c` in context output format while ignoring case differences, you might run **svn diff --diff-cmd /usr/bin/diff --extensions '-i' foo.c**.

Undoing Working Changes

Suppose while viewing the output of **svn diff** you determine that all the changes you made to a particular file are mistakes. Maybe you shouldn't have changed the file at all, or perhaps it would be easier to make different changes starting from scratch.

This is a perfect opportunity to use **svn revert**:

```
$ svn revert README
Reverted 'README'
```

Subversion reverts the file to its pre-modified state by overwriting it with the cached “pristine” copy from the `.svn` area. But also note that **svn revert** can undo *any* scheduled operations—for example, you might decide that you don't want to add a new file after all:

```
$ svn status foo
?      foo

$ svn add foo
A         foo

$ svn revert foo
Reverted 'foo'

$ svn status foo
?      foo
```



svn revert *ITEM* has exactly the same effect as deleting *ITEM* from your working copy and then running **svn update -r BASE** *ITEM*. However, if you're reverting a file, **svn revert** has one very noticeable difference—it doesn't have to communicate with the repository to restore your file.

Or perhaps you mistakenly removed a file from version control:

```
$ svn status README
       README

$ svn delete README
D         README

$ svn revert README
Reverted 'README'

$ svn status README
       README
```

Resolve Conflicts (Merging Others' Changes)

We've already seen how **svn status -u** can predict conflicts. Suppose you run **svn update** and

some interesting things occur:

```
$ svn update
U  INSTALL
G  README
C  bar.c
Updated to revision 46.
```

The U and G codes are no cause for concern; those files cleanly absorbed changes from the repository. The files marked with U contained no local changes but were Updated with changes from the repository. The G stands for merGed, which means that the file had local changes to begin with, but the changes coming from the repository didn't overlap with the local changes.

But the C stands for conflict. This means that the changes from the server overlapped with your own, and now you have to manually choose between them.

Whenever a conflict occurs, three things typically occur to assist you in noticing and resolving that conflict:

- Subversion prints a C during the update, and remembers that the file is in a state of conflict.
- If Subversion considers the file to be mergeable, it places *conflict markers*—special strings of text which delimit the "sides" of the conflict—into the file to visibly demonstrate the overlapping areas. (Subversion uses the `svn:mime-type` property to decide if a file is capable of contextual, line-based merging. See the section called "File Content Type" to learn more.)
- For every conflicted file, Subversion places three extra unversioned files in your working copy:

  `filename.mine`
  This is your file as it existed in your working copy before you updated your working copy—that is, without conflict markers. This file has only your latest changes in it. (If Subversion considers the file to be unmergeable, then the `.mine` file isn't created, since it would be identical to the working file.)

  `filename.rOLDREV`
  This is the file that was the BASE revision before you updated your working copy. That is, the file that you checked out before you made your latest edits.

  `filename.rNEWREV`
  This is the file that your Subversion client just received from the server when you updated your working copy. This file corresponds to the HEAD revision of the repository.

  Here OLDREV is the revision number of the file in your `.svn` directory and NEWREV is the revision number of the repository HEAD.

For example, Sally makes changes to the file `sandwich.txt` in the repository. Harry has just changed the file in his working copy and checked it in. Sally updates her working copy before checking in and she gets a conflict:

```
$ svn update
C  sandwich.txt
Updated to revision 2.
$ ls -1
sandwich.txt
```

```
sandwich.txt.mine
sandwich.txt.r1
sandwich.txt.r2
```

At this point, Subversion will *not* allow you to commit the file `sandwich.txt` until the three temporary files are removed.

```
$ svn commit -m "Add a few more things"
svn: Commit failed (details follow):
svn: Aborting commit: '/home/sally/svn-work/sandwich.txt' remains in conflict
```

If you get a conflict, you need to do one of three things:

- Merge the conflicted text “by hand” (by examining and editing the conflict markers within the file).
- Copy one of the temporary files on top of your working file.
- Run **svn revert <filename>** to throw away all of your local changes.

Once you've resolved the conflict, you need to let Subversion know by running **svn resolved**. This removes the three temporary files and Subversion no longer considers the file to be in a state of conflict.[4]

```
$ svn resolved sandwich.txt
Resolved conflicted state of 'sandwich.txt'
```

Merging Conflicts by Hand

Merging conflicts by hand can be quite intimidating the first time you attempt it, but with a little practice, it can become as easy as falling off a bike.

Here's an example. Due to a miscommunication, you and Sally, your collaborator, both edit the file `sandwich.txt` at the same time. Sally commits her changes, and when you go to update your working copy, you get a conflict and you're going to have to edit `sandwich.txt` to resolve the conflicts. First, let's take a look at the file:

```
$ cat sandwich.txt
Top piece of bread
Mayonnaise
Lettuce
Tomato
Provolone
<<<<<<< .mine
Salami
Mortadella
Prosciutto
=======
Sauerkraut
Grilled Chicken
```

[4] You can always remove the temporary files yourself, but would you really want to do that when Subversion can do it for you? We didn't think so.

```
>>>>>>> .r2
Creole Mustard
Bottom piece of bread
```

The strings of less-than signs, equal signs, and greater-than signs are conflict markers, and are not part of the actual data in conflict. You generally want to ensure that those are removed from the file before your next commit. The text between the first two sets of markers is composed of the changes you made in the conflicting area:

```
<<<<<<< .mine
Salami
Mortadella
Prosciutto
=======
```

The text between the second and third sets of conflict markers is the text from Sally's commit:

```
=======
Sauerkraut
Grilled Chicken
>>>>>>> .r2
```

Usually you won't want to just delete the conflict markers and Sally's changes—she's going to be awfully surprised when the sandwich arrives and it's not what she wanted. So this is where you pick up the phone or walk across the office and explain to Sally that you can't get sauerkraut from an Italian deli.[5] Once you've agreed on the changes you will check in, edit your file and remove the conflict markers.

```
Top piece of bread
Mayonnaise
Lettuce
Tomato
Provolone
Salami
Mortadella
Prosciutto
Creole Mustard
Bottom piece of bread
```

Now run **svn resolved**, and you're ready to commit your changes:

```
$ svn resolved sandwich.txt
$ svn commit -m "Go ahead and use my sandwich, discarding Sally's edits."
```

Note that **svn resolved**, unlike most of the other commands we deal with in this chapter, requires an argument. In any case, you want to be careful and only run **svn resolved** when you're certain that you've fixed the conflict in your file—once the temporary files are removed, Subversion will let you commit the file even if it still contains conflict markers.

If you ever get confused while editing the conflicted file, you can always consult the three files

[5] And if you ask them for it, they may very well ride you out of town on a rail.

that Subversion creates for you in your working copy—including your file as it was before you updated. You can even use a third-party interactive merging tool to examine those three files.

Copying a File Onto Your Working File

If you get a conflict and decide that you want to throw out your changes, you can merely copy one of the temporary files created by Subversion over the file in your working copy:

```
$ svn update
C  sandwich.txt
Updated to revision 2.
$ ls sandwich.*
sandwich.txt  sandwich.txt.mine  sandwich.txt.r2  sandwich.txt.r1
$ cp sandwich.txt.r2 sandwich.txt
$ svn resolved sandwich.txt
```

Punting: Using svn revert

If you get a conflict, and upon examination decide that you want to throw out your changes and start your edits again, just revert your changes:

```
$ svn revert sandwich.txt
Reverted 'sandwich.txt'
$ ls sandwich.*
sandwich.txt
```

Note that when you revert a conflicted file, you don't have to run **svn resolved**.

Commit Your Changes

Finally! Your edits are finished, you've merged all changes from the server, and you're ready to commit your changes to the repository.

The **svn commit** command sends all of your changes to the repository. When you commit a change, you need to supply a *log message*, describing your change. Your log message will be attached to the new revision you create. If your log message is brief, you may wish to supply it on the command line using the `--message` (or `-m`) option:

```
$ svn commit -m "Corrected number of cheese slices."
Sending        sandwich.txt
Transmitting file data .
Committed revision 3.
```

However, if you've been composing your log message as you work, you may want to tell Subversion to get the message from a file by passing the filename with the `--file (-F)` option:

```
$ svn commit -F logmsg
Sending        sandwich.txt
Transmitting file data .
Committed revision 4.
```

If you fail to specify either the `--message` or `--file` option, then Subversion will automatically launch your favorite editor (see the `editor-cmd` section in the section called “Config”) for composing a log message.



If you're in your editor writing a commit message and decide that you want to cancel your commit, you can just quit your editor without saving changes. If you've already saved your commit message, simply delete the text, save again, then abort.

```
$ svn commit
Waiting for Emacs...Done

Log message unchanged or not specified
a)bort, c)ontinue, e)dit
a
$
```

The repository doesn't know or care if your changes make any sense as a whole; it only checks to make sure that nobody else has changed any of the same files that you did when you weren't looking. If somebody *has* done that, the entire commit will fail with a message informing you that one or more of your files is out-of-date:

```
$ svn commit -m "Add another rule"
Sending        rules.txt
svn: Commit failed (details follow):
svn: Your file or directory 'sandwich.txt' is probably out-of-date
…
```

(The exact wording of this error message depends on the network protocol and server you're using, but the idea is the same in all cases.)

At this point, you need to run **svn update**, deal with any merges or conflicts that result, and attempt your commit again.

That covers the basic work cycle for using Subversion. There are many other features in Subversion that you can use to manage your repository and working copy, but most of your day-to-day use of Subversion will involve only the commands that we've discussed so far in this chapter. We will, however, cover a few more commands that you'll use fairly often.

Examining History

Your Subversion repository is like a time machine. It keeps a record of every change ever committed, and allows you to explore this history by examining previous versions of files and directories as well as the metadata that accompanies them. With a single Subversion command, you can check out the repository (or restore an existing working copy) exactly as it was at any date or revision number in the past. However, sometimes you just want to *peer into* the past instead of *going into* the past.

There are several commands that can provide you with historical data from the repository:

svn log
: Shows you broad information: log messages with date and author information attached to

revisions, and which paths changed in each revision.

svn diff
Shows line-level details of a particular change.

svn cat
Retrieves a file as it existed in a particular revision number and display it on your screen.

svn list
Displays the files in a directory for any given revision.

Generating a list of historical changes

To find information about the history of a file or directory, use the **svn log** command. **svn log** will provide you with a record of who made changes to a file or directory, at what revision it changed, the time and date of that revision, and, if it was provided, the log message that accompanied the commit.

```
$ svn log
------------------------------------------------------------------------
r3 | sally | Mon, 15 Jul 2002 18:03:46 -0500 | 1 line

Added include lines and corrected # of cheese slices.
------------------------------------------------------------------------
r2 | harry | Mon, 15 Jul 2002 17:47:57 -0500 | 1 line

Added main() methods.
------------------------------------------------------------------------
r1 | sally | Mon, 15 Jul 2002 17:40:08 -0500 | 1 line

Initial import
------------------------------------------------------------------------
```

Note that the log messages are printed in *reverse chronological order* by default. If you wish to see a different range of revisions in a particular order, or just a single revision, pass the `--revision (-r)` option:

```
$ svn log -r 5:19    # shows logs 5 through 19 in chronological order

$ svn log -r 19:5    # shows logs 5 through 19 in reverse order

$ svn log -r 8       # shows log for revision 8
```

You can also examine the log history of a single file or directory. For example:

```
$ svn log foo.c
…
$ svn log http://foo.com/svn/trunk/code/foo.c
…
```

These will display log messages *only* for those revisions in which the working file (or URL) changed.

If you want even more information about a file or directory, **svn log** also takes a `--verbose`

(-v) option. Because Subversion allows you to move and copy files and directories, it is important to be able to track path changes in the filesystem, so in verbose mode, **svn log** will include a list of changed paths in a revision in its output:

```
$ svn log -r 8 -v
------------------------------------------------------------------------
r8 | sally | 2002-07-14 08:15:29 -0500 | 1 line
Changed paths:
M /trunk/code/foo.c
M /trunk/code/bar.h
A /trunk/code/doc/README

Frozzled the sub-space winch.

------------------------------------------------------------------------
```

svn log also takes a `--quiet` (`-q`) option, which suppresses the body of the log message. When combined with `--verbose`, it gives just the names of the changed files.

Why Does svn log Give Me an Empty Response?

After working with Subversion for a bit, most users will come across something like this:

```
$ svn log -r 2
------------------------------------------------------------------------
$
```

At first glance, this seems like an error. But recall that while revisions are repository-wide, **svn log** operates on a path in the repository. If you supply no path, Subversion uses the current working directory as the default target. As a result, if you're operating in a subdirectory of your working copy and attempt to see the log of a revision in which neither that directory nor any of its children was changed, Subversion will show you an empty log. If you want to see what changed in that revision, try pointing **svn log** directly at the topmost URL of your repository, as in **svn log -r 2 http://svn.collab.net/repos/svn**.

Examining the details of historical changes

We've already seen **svn diff** before—it displays file differences in unified diff format; it was used to show the local modifications made to our working copy before committing to the repository.

In fact, it turns out that there are *three* distinct uses of **svn diff**:

- Examining local changes
- Comparing your working copy to the repository
- Comparing repository to repository

Examining Local Changes

As we've seen, invoking **svn diff** with no options will compare your working files to the cached

“pristine” copies in the `.svn` area:

```
$ svn diff
Index: rules.txt
===================================================================
--- rules.txt (revision 3)
+++ rules.txt (working copy)
@@ -1,4 +1,5 @@
 Be kind to others
 Freedom = Responsibility
 Everything in moderation
-Chew with your mouth open
+Chew with your mouth closed
+Listen when others are speaking
$
```

Comparing Working Copy to Repository

If a single `--revision` (`-r`) number is passed, then your working copy is compared to the specified revision in the repository.

```
$ svn diff -r 3 rules.txt
Index: rules.txt
===================================================================
--- rules.txt (revision 3)
+++ rules.txt (working copy)
@@ -1,4 +1,5 @@
 Be kind to others
 Freedom = Responsibility
 Everything in moderation
-Chew with your mouth open
+Chew with your mouth closed
+Listen when others are speaking
$
```

Comparing Repository to Repository

If two revision numbers, separated by a colon, are passed via `--revision (-r)`, then the two revisions are directly compared.

```
$ svn diff -r 2:3 rules.txt
Index: rules.txt
===================================================================
--- rules.txt (revision 2)
+++ rules.txt (revision 3)
@@ -1,4 +1,4 @@
 Be kind to others
-Freedom = Chocolate Ice Cream
+Freedom = Responsibility
 Everything in moderation
 Chew with your mouth open
$
```

A more convenient way of comparing a revision to the previous revision is to use the `--change (-c)`:

```
$ svn diff -c 3 rules.txt
Index: rules.txt
===================================================================
--- rules.txt (revision 2)
+++ rules.txt (revision 3)
@@ -1,4 +1,4 @@
 Be kind to others
-Freedom = Chocolate Ice Cream
+Freedom = Responsibility
 Everything in moderation
 Chew with your mouth open
$
```

Lastly, you can compare repository revisions even when you don't have a working copy on your local machine, just by including the appropriate URL on the command line:

```
$ svn diff -c 5 http://svn.example.com/repos/example/trunk/text/rules.txt
…
$
```

Browsing the repository

Using **svn cat** and **svn list**, you can view various revisions of files and directories without changing the working revision of your working copy. In fact, you don't even need a working copy to use either one.

svn cat

If you want to examine an earlier version of a file and not necessarily the differences between two files, you can use **svn cat**:

```
$ svn cat -r 2 rules.txt
Be kind to others
Freedom = Chocolate Ice Cream
Everything in moderation
Chew with your mouth open
$
```

You can also redirect the output directly into a file:

```
$ svn cat -r 2 rules.txt > rules.txt.v2
$
```

svn list

The **svn list** command shows you what files are in a repository directory without actually downloading the files to your local machine:

```
$ svn list http://svn.collab.net/repos/svn
README
branches/
clients/
```

```
tags/
trunk/
```

If you want a more detailed listing, pass the `--verbose (-v)` flag to get output like this:

```
$ svn list -v http://svn.collab.net/repos/svn
  20620 harry            1084 Jul 13  2006 README
  23339 harry                 Feb 04 01:40 branches/
  21282 sally                 Aug 27 09:41 developer-resources/
  23198 harry                 Jan 23 17:17 tags/
  23351 sally                 Feb 05 13:26 trunk/
```

The columns tell you the revision at which the file or directory was last modified, the user who modified it, the size if it is a file, the date it was last modified, and the item's name.



The **svn list** with no arguments defaults to the *repository URL* of the current working directory, *not* the local working copy directory. After all, if you wanted a listing of your local directory, you could use just plain **ls** (or any reasonable non-Unixy equivalent).

Fetching older repository snapshots

In addition to all of the above commands, you can use **svn update** and **svn checkout** with the `--revision` option to take an entire working copy “back in time” [6]:

```
$ svn checkout -r 1729 # Checks out a new working copy at r1729
…
$ svn update -r 1729 # Updates an existing working copy to r1729
…
```



Many Subversion newcomers attempt to use the above **svn update** example to “undo” committed changes, but this won't work as you can't commit changes that you obtain from backdating a working copy if the changed files have newer revisions. See the section called “Resurrecting Deleted Items” for a description of how to “undo” a commit.

Lastly, if you're building a release and wish to bundle up your files from Subversion but don't want those pesky `.svn` directories in the way, then you can use **svn export** to create a local copy of all or part of your repository sans `.svn` directories. As with **svn update** and **svn checkout**, you can also pass the `--revision` option to **svn export**:

```
$ svn export http://svn.example.com/svn/repos1 # Exports latest revision
…
$ svn export http://svn.example.com/svn/repos1 -r 1729
# Exports revision r1729
…
```

[6] See? We told you that Subversion was a time machine.

Sometimes You Just Need to Clean Up

When Subversion modifies your working copy (or any information within `.svn`), it tries to do so as safely as possible. Before changing the working copy, Subversion writes its intentions to a log file. Next it executes the commands in the log file to apply the requested change, holding a lock on the relevant part of the working copy while it works—to prevent other Subversion clients from accessing the working copy in mid-change. Finally, Subversion removes the log file. Architecturally, this is similar to a journaled filesystem. If a Subversion operation is interrupted (if the process is killed, or if the machine crashes, for example), the log files remain on disk. By re-executing the log files, Subversion can complete the previously started operation, and your working copy can get itself back into a consistent state.

And this is exactly what **svn cleanup** does: it searches your working copy and runs any leftover logs, removing working copy locks in the process. If Subversion ever tells you that some part of your working copy is “locked”, then this is the command that you should run. Also, **svn status** will display an `L` next to locked items:

```
$ svn status
  L    somedir
M      somedir/foo.c

$ svn cleanup
$ svn status
M      somedir/foo.c
```

Don't confuse these working copy locks with the ordinary locks that Subversion users create when using the “lock-modify-unlock” model of concurrent version control; see The three meanings of “lock” for clarification.

Summary

Now we've covered most of the Subversion client commands. Notable exceptions are those dealing with branching and merging (see Chapter 4, *Branching and Merging*) and properties (see the section called “Properties”). However, you may want to take a moment to skim through Chapter 9, *Subversion Complete Reference* to get an idea of all the many different commands that Subversion has—and how you can use them to make your work easier.

Chapter 3. Advanced Topics

If you've been reading this book chapter by chapter, from start to finish, you should by now have acquired enough knowledge to use the Subversion client to perform the most common version control operations. You understand how to check out a working copy from a Subversion repository. You are comfortable with submitting and receiving changes using the **svn commit** and **svn update** functions. You've probably even developed a reflex which causes you to run the **svn status** command almost unconsciously. For all intents and purposes, you are ready to use Subversion in a typical environment.

But the Subversion feature set doesn't stop at “common version control operations”. It has other bits of functionality besides just communicating file and directory changes to and from a central repository.

This chapter highlights some of Subversion's features that, while important, aren't part of the typical user's daily routine. It assumes that you are familiar with Subversion's basic file and directory versioning capabilities. If you aren't, you'll want to first read Chapter 1, *Fundamental Concepts* and Chapter 2, *Basic Usage*. Once you've mastered those basics and consumed this chapter, you'll be a Subversion power-user!

Revision Specifiers

As you saw in the section called “Revisions”, revision numbers in Subversion are pretty straightforward—integers that keep getting larger as you commit more changes to your versioned data. Still, it doesn't take long before you can no longer remember exactly what happened in each and every revision. Fortunately, the typical Subversion workflow doesn't often demand that you supply arbitrary revisions to the Subversion operations you perform. For operations that *do* require a revision specifier, you generally supply a revision number that you saw in a commit email, in the output of some other Subversion operation, or in some other context that would give meaning to that particular number.

But occasionally, you need to pinpoint a moment in time for which you don't already have a revision number memorized or handy. So besides the integer revision numbers, **svn** allows as input some additional forms of revision specifiers—*revision keywords*, and revision dates.



The various forms of Subversion revision specifiers can be mixed and matched when used to specify revision ranges. For example, you can use `-r` *REV1*`:`*REV2* where *REV1* is a revision keyword and *REV2* is a revision number, or where *REV1* is a date and *REV2* is a revision keyword, and so on. The individual revision specifiers are independently evaluated, so you can put whatever you want on the opposite sides of that colon.

Revision Keywords

The Subversion client understands a number of revision keywords. These keywords can be used instead of integer arguments to the `--revision (-r)` switch, and are resolved into specific revision numbers by Subversion:

HEAD
: The latest (or “youngest”) revision in the repository.

BASE
: The revision number of an item in a working copy. If the item has been locally modified, the "BASE version" refers to the way the item appears without those local modifications.

COMMITTED
: The most recent revision prior to, or equal to, `BASE`, in which an item changed.

PREV
: The revision immediately *before* the last revision in which an item changed. Technically, this boils down to `COMMITTED-1`.

As can be derived from their descriptions, the `PREV`, `BASE`, and `COMMITTED` revision keywords are used only when referring to a working copy path—they don't apply to repository URLs. `HEAD`, on the other hand, can be used in conjunction with both of these path types.

Here are some examples of revision keywords in action:

```
$ svn diff -r PREV:COMMITTED foo.c
# shows the last change committed to foo.c

$ svn log -r HEAD
# shows log message for the latest repository commit

$ svn diff -r HEAD
# compares your working copy (with all of its local changes) to the
# latest version of that tree in the repository

$ svn diff -r BASE:HEAD foo.c
# compares the unmodified version of foo.c with the latest version of
# foo.c in the repository

$ svn log -r BASE:HEAD
# shows all commit logs for the current versioned directory since you
# last updated

$ svn update -r PREV foo.c
# rewinds the last change on foo.c, decreasing foo.c's working revision

$ svn diff -r BASE:14 foo.c
# compares the unmodified version of foo.c with the way foo.c looked
# in revision 14
```

Revision Dates

Revision numbers reveal nothing about the world outside the version control system, but sometimes you need to correlate a moment in real time with a moment in version history. To facilitate this, the `--revision (-r)` option can also accept as input date specifiers wrapped in curly braces (`{` and `}`). Subversion accepts the standard ISO-8601 date and time formats, plus a few others. Here are some examples. (Remember to use quotes around any date that contains spaces.)

```
$ svn checkout -r {2006-02-17}
$ svn checkout -r {15:30}
$ svn checkout -r {15:30:00.200000}
$ svn checkout -r {"2006-02-17 15:30"}
$ svn checkout -r {"2006-02-17 15:30 +0230"}
```

```
$ svn checkout -r {2006-02-17T15:30}
$ svn checkout -r {2006-02-17T15:30Z}
$ svn checkout -r {2006-02-17T15:30-04:00}
$ svn checkout -r {20060217T1530}
$ svn checkout -r {20060217T1530Z}
$ svn checkout -r {20060217T1530-0500}
…
```

When you specify a date, Subversion resolves that date to the most recent revision of the repository as of that date, and then continues to operate against that resolved revision number:

```
$ svn log -r {2006-11-28}
------------------------------------------------------------------------
r12 | ira | 2006-11-27 12:31:51 -0600 (Mon, 27 Nov 2006) | 6 lines
…
```

Is Subversion a Day Early?

If you specify a single date as a revision without specifying a time of day (for example `2006-11-27`), you may think that Subversion should give you the last revision that took place on the 27th of November. Instead, you'll get back a revision from the 26th, or even earlier. Remember that Subversion will find the *most recent revision of the repository* as of the date you give. If you give a date without a timestamp, like `2006-11-27`, Subversion assumes a time of 00:00:00, so looking for the most recent revision won't return anything on the day of the 27th.

If you want to include the 27th in your search, you can either specify the 27th with the time (`{"2006-11-27 23:59"}`), or just specify the next day (`{2006-11-28}`).

You can also use a range of dates. Subversion will find all revisions between both dates, inclusive:

```
$ svn log -r {2006-11-20}:{2006-11-29}
…
```



Since the timestamp of a revision is stored as an unversioned, modifiable property of the revision (see the section called “Properties”, revision timestamps can be changed to represent complete falsifications of true chronology, or even removed altogether. Subversion's ability to correctly convert revision dates into real revision numbers depends on revision datestamps maintaining a sequential ordering—the younger the revision, the younger its timestamp. If this ordering isn't maintained, you will likely find that trying to use dates to specify revision ranges in your repository doesn't always return the data you might have expected.

Properties

We've already covered in detail how Subversion stores and retrieves various versions of files and directories in its repository. Whole chapters have been devoted to this most fundamental piece of functionality provided by the tool. And if the versioning support stopped there, Subversion would still be complete from a version control perspective.

But it doesn't stop there.

In addition to versioning your directories and files, Subversion provides interfaces for adding, modifying, and removing versioned metadata on each of your versioned directories and files. We refer to this metadata as *properties*, and they can be thought of as two-column tables that map property names to arbitrary values attached to each item in your working copy. Generally speaking, the names and values of the properties can be whatever you want them to be, with the constraint that the names must be human-readable text. And the best part about these properties is that they, too, are versioned, just like the textual contents of your files. You can modify, commit, and revert property changes as easily as you can file content changes. And the sending and receiving of property changes occurs as part of your typical commit and update operations—you don't have to change your basic processes to accommodate them.



Subversion has reserved the set of properties whose names begin with `svn:` as its own. While there are only a handful of such properties in use today, you should avoid creating custom properties for your own needs whose names begin with this prefix. Otherwise, you run the risk that a future release of Subversion will grow support for a feature or behavior driven by a property of the same name but with perhaps an entirely different interpretation.

Properties show up elsewhere in Subversion, too. Just as files and directories may have arbitrary property names and values attached to them, each revision as a whole may have arbitrary properties attached to it. The same constraints apply—human-readable names and anything-you-want binary values. The main difference is that revision properties are not versioned. In other words, if you change the value of, or delete, a revision property, there's no way within the scope of Subversion's functionality to recover the previous value.

Subversion has no particular policy regarding the use of properties. It asks only that you not use property names that begin with the prefix `svn:`. That's the namespace that it sets aside for its own use. And Subversion does, in fact, use properties, both the versioned and unversioned variety. Certain versioned properties have special meaning or effects when found on files and directories, or house a particular bit of information about the revisions on which they are found. Certain revision properties are automatically attached to revisions by Subversion's commit process, and carry information about the revision. Most of these properties are mentioned elsewhere in this or other chapters as part of the more general topics to which they are related. For an exhaustive list of Subversion's pre-defined properties, see the section called “Subversion properties”.

In this section, we will examine the utility—both to users of Subversion, and to Subversion itself—of property support. You'll learn about the property-related **svn** subcommands, and how property modifications affect your normal Subversion workflow.

Why Properties?

Just as Subversion uses properties to store extra information about the files, directories, and revisions that it contains, you might also find properties to be of similar use. You might find it useful to have a place close to your versioned data to hang custom metadata about that data.

Say you wish to design a website that houses many digital photos, and displays them with captions and a datestamp. Now, your set of photos is constantly changing, so you'd like to have as much of this site automated as possible. These photos can be quite large, so as is common with sites of this nature, you want to provide smaller thumbnail images to your site visitors.

Now, you can get this functionality using traditional files. That is, you can have your `image123.jpg` and an `image123-thumbnail.jpg` side-by-side in a directory. Or if you want

to keep the filenames the same, you might have your thumbnails in a different directory, like `thumbnails/image123.jpg`. You can also store your captions and datestamps in a similar fashion, again separated from the original image file. But the problem here is that your collection of files grows in multiples with each new photo added to the site.

Now consider the same website deployed in a way that makes use of Subversion's file properties. Imagine having a single image file, `image123.jpg`, and then properties set on that file named `caption`, `datestamp`, and even `thumbnail`. Now your working copy directory looks much more manageable—in fact, it looks to the casual browser like there are nothing but image files in it. But your automation scripts know better. They know that they can use **svn** (or better yet, they can use the Subversion language bindings—see the section called “Using the APIs”) to dig out the extra information that your site needs to display without having to read an index file or play path manipulation games.

Custom revision properties are also frequently used. One common such use is a property whose value contains an issue tracker ID with which the revision is associated, perhaps because the change made in that revision fixes a bug filed in the tracker issue with that ID. Other uses include hanging more friendly names on the revision—it might be hard to remember that revision 1935 was a fully tested revision. But if there's, say, a `test-results` property on that revision with a value `all passing`, that's meaningful information to have.

Searchability (or, Why *Not* Properties)

For all their utility, Subversion properties—or, more accurately, the available interfaces to them—have a major shortcoming: while it is a simple matter to *set* a custom property, *finding* that property later is whole different ball of wax.

Trying to locate a custom revision property generally involves performing a linear walk across all the revisions of the repository, asking of each revision, "Do you have the property I'm looking for?" Trying to find a custom versioned property is painful, too, and often involves a recursive **svn propget** across an entire working copy. In your situation, that might not be as bad as a linear walk across all revisions. But it certainly leaves much to be desired in terms of both performance and likelihood of success, especially if the scope of your search would require a working copy from the root of your repository.

For this reason, you might choose—especially in the revision property use-case—to simply add your metadata to the revision's log message, using some policy-driven (and perhaps programmatically-enforced) formatting that is designed to be quickly parsed from the output of **svn log**. It is quite common to see in Subversion log messages the likes of:

```
Issue(s): IZ2376, IZ1919
Reviewed by:  sally

This fixes a nasty segfault in the wort frabbing process
…
```

But here again lies some misfortune. Subversion doesn't yet provide a log message templating mechanism, which would go a long way toward helping users be consistent with the formatting of their log-embedded revision metadata.

Manipulating Properties

The **svn** command affords a few ways to add or modify file and directory properties. For properties with short, human-readable values, perhaps the simplest way to add a new property is to

specify the property name and value on the command line of the **propset** subcommand.

```
$ svn propset copyright '(c) 2006 Red-Bean Software' calc/button.c
property 'copyright' set on 'calc/button.c'
$
```

But we've been touting the flexibility that Subversion offers for your property values. And if you are planning to have a multi-line textual, or even binary, property value, you probably do not want to supply that value on the command line. So the **propset** subcommand takes a `--file (-F)` option for specifying the name of a file which contains the new property value.

```
$ svn propset license -F /path/to/LICENSE calc/button.c
property 'license' set on 'calc/button.c'
$
```

There are some restrictions on the names you can use for properties. A property name must start with a letter, a colon (`:`), or an underscore (_); after that, you can also use digits, hyphens (`-`), and periods (`.`). [1]

In addition to the **propset** command, the **svn** program supplies the **propedit** command. This command uses the configured editor program (see the section called “Config”) to add or modify properties. When you run the command, **svn** invokes your editor program on a temporary file that contains the current value of the property (or which is empty, if you are adding a new property). Then, you just modify that value in your editor program until it represents the new value you wish to store for the property, save the temporary file, and then exit the editor program. If Subversion detects that you've actually changed the existing value of the property, it will accept that as the new property value. If you exit your editor without making any changes, no property modification will occur:

```
$ svn propedit copyright calc/button.c  ### exit the editor without changes
No changes to property 'copyright' on 'calc/button.c'
$
```

We should note that, as with other **svn** subcommands, those related to properties can act on multiple paths at once. This enables you to modify properties on whole sets of files with a single command. For example, we could have done:

```
$ svn propset copyright '(c) 2006 Red-Bean Software' calc/*
property 'copyright' set on 'calc/Makefile'
property 'copyright' set on 'calc/button.c'
property 'copyright' set on 'calc/integer.c'
…
$
```

All of this property adding and editing isn't really very useful if you can't easily get the stored property value. So the **svn** program supplies two subcommands for displaying the names and values of properties stored on files and directories. The **svn proplist** command will list the names of properties that exist on a path. Once you know the names of the properties on the node, you can request their values individually using **svn propget**. This command will, given a property name and a path (or set of paths), print the value of the property to the standard output stream.

[1] If you're familiar with XML, this is pretty much the ASCII subset of the syntax for XML "Name".

```
$ svn proplist calc/button.c
Properties on 'calc/button.c':
  copyright
  license
$ svn propget copyright calc/button.c
(c) 2006 Red-Bean Software
```

There's even a variation of the **proplist** command that will list both the name and value of all of the properties. Simply supply the `--verbose` (`-v`) option.

```
$ svn proplist -v calc/button.c
Properties on 'calc/button.c':
  copyright : (c) 2006 Red-Bean Software
  license : ================================================================
Copyright (c) 2006 Red-Bean Software.  All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions, and the recipe for Fitz's famous
red-beans-and-rice.
…
```

The last property-related subcommand is **propdel**. Since Subversion allows you to store properties with empty values, you can't remove a property altogether using **propedit** or **propset**. For example, this command will *not* yield the desired effect:

```
$ svn propset license '' calc/button.c
property 'license' set on 'calc/button.c'
$ svn proplist -v calc/button.c
Properties on 'calc/button.c':
  copyright : (c) 2006 Red-Bean Software
  license :
$
```

You need to use the **propdel** subcommand to delete properties altogether. The syntax is similar to the other property commands:

```
$ svn propdel license calc/button.c
property 'license' deleted from 'calc/button.c'.
$ svn proplist -v calc/button.c
Properties on 'calc/button.c':
  copyright : (c) 2006 Red-Bean Software
$
```

Remember those unversioned revision properties? You can modify those, too, using the same **svn** subcommands that we just described. Simply add the `--revprop` command-line parameter, and specify the revision whose property you wish to modify. Since revisions are global, you don't need to specify a target path to these property-related commands so long as you are positioned in a working copy of the repository whose revision property you wish to modify. Otherwise, you can simply provide the URL of any path in the repository of interest (including the repository's root URL). For example, you might want to replace the commit log message of an

existing revision. [2] If your current working directory is part of a working copy of your repository, you can simply run the **svn propset** command with no target path:

```
$ svn propset svn:log '* button.c: Fix a compiler warning.' -r11 --revprop
property 'svn:log' set on repository revision '11'
$
```

But even if you haven't checked out a working copy from that repository, you can still affect the property change by providing the repository's root URL:

```
$ svn propset svn:log '* button.c: Fix a compiler warning.' -r11 --revprop \
            http://svn.example.com/repos/project
property 'svn:log' set on repository revision '11'
$
```

Note that the ability to modify these unversioned properties must be explicitly added by the repository administrator (see the section called “Commit Log Message Correction”). That's because the properties aren't versioned, so you run the risk of losing information if you aren't careful with your edits. The repository administrator can set up methods to protect against this loss, and by default, modification of unversioned properties is disabled.



Users should, where possible, use **svn propedit** instead of **svn propset**. While the end result of the commands is identical, the former will allow them to see the current value of the property they are about to change, which helps them to verify that they are, in fact, making the change they think they are making. This is especially true when modifying unversioned revision properties. Also, it is significantly easier to modify multiline property values in a text editor than at the command line.

Properties and the Subversion Workflow

Now that you are familiar with all of the property-related **svn** subcommands, let's see how property modifications affect the usual Subversion workflow. As we mentioned earlier, file and directory properties are versioned, just like your file contents. As a result, Subversion provides the same opportunities for merging—cleanly or with conflicts—someone else's modifications into your own.

And as with file contents, your property changes are local modifications, only made permanent when you commit them to the repository with **svn commit**. Your property changes can be easily unmade, too—the **svn revert** command will restore your files and directories to their unedited states—contents, properties, and all. Also, you can receive interesting information about the state of your file and directory properties by using the **svn status** and **svn diff** commands.

```
$ svn status calc/button.c
 M     calc/button.c
$ svn diff calc/button.c
Property changes on: calc/button.c
___________________________________________________________________
Name: copyright
   + (c) 2006 Red-Bean Software
```

[2] Fixing spelling errors, grammatical gotchas, and “just-plain-wrongness” in commit log messages is perhaps the most common use case for the `--revprop` option.

```
$
```

Notice how the **status** subcommand displays `M` in the second column instead of the first. That is because we have modified the properties on `calc/button.c`, but not its textual contents. Had we changed both, we would have seen `M` in the first column, too (see the section called “See an overview of your changes”).

Property Conflicts

As with file contents, local property modifications can conflict with changes committed by someone else. If you update your working copy directory and receive property changes on a versioned object that clash with your own, Subversion will report that the object is in a conflicted state.

```
% svn update calc
M  calc/Makefile.in
 C calc/button.c
Updated to revision 143.
$
```

Subversion will also create, in the same directory as the conflicted object, a file with a `.prej` extension which contains the details of the conflict. You should examine the contents of this file so you can decide how to resolve the conflict. Until the conflict is resolved, you will see a `C` in the second column of **svn status** output for that object, and attempts to commit your local modifications will fail.

```
$ svn status calc
 C     calc/button.c
?      calc/button.c.prej
$ cat calc/button.c.prej
prop 'linecount': user set to '1256', but update set to '1301'.
$
```

To resolve property conflicts, simply ensure that the conflicting properties contain the values that they should, and then use the **svn resolved** command to alert Subversion that you have manually resolved the problem.

You might also have noticed the non-standard way that Subversion currently displays property differences. You can still run **svn diff** and redirect the output to create a usable patch file. The **patch** program will ignore property patches—as a rule, it ignores any noise it can't understand. This does, unfortunately, mean that to fully apply a patch generated by **svn diff**, any property modifications will need to be applied by hand.

Automatic Property Setting

Properties are a powerful feature of Subversion, acting as key components of many Subversion features discussed elsewhere in this and other chapters—textual diff and merge support, keyword substitution, newline translation, etc. But to get the full benefit of properties, they must be set on the right files and directories. Unfortunately, that step can be easily forgotten in the routine of things, especially since failing to set a property doesn't usually result in an obvious error (at least compared to, say, failing to add a file to version control). To help your properties get applied to the places that need them, Subversion provides a couple of simple but useful

features.

Whenever you introduce a file to version control using the **svn add** or **svn import** commands, Subversion tries to assist by setting some common file properties automatically. First, on operating systems whose filesystems support an execute permission bit, Subversion will automatically set the `svn:executable` property on newly added or imported files whose execute bit is enabled. (See the section called “File Executability” for more about this property.) Secondly, it runs a very basic heuristic to determine if that file contains human-readable content. If not, Subversion will automatically set the `svn:mime-type` property on that file to `application/octet-stream` (the generic “this is a collection of bytes” MIME type). Of course, if Subversion guesses incorrectly, or if you wish to set the `svn:mime-type` property to something more precise—perhaps `image/png` or `application/x-shockwave-flash`—you can always remove or edit that property. (For more on Subversion's use of MIME types, see the section called “File Content Type”.)

Subversion also provides, via its runtime configuration system (see the section called “Runtime Configuration Area”), a more flexible automatic property setting feature which allows you to create mappings of filename patterns to property names and values. Once again, these mappings affect adds and imports, and can not only override the default MIME type decision made by Subversion during those operations, but can also set additional Subversion or custom properties, too. For example, you might create a mapping that says that any time you add JPEG files—ones whose names match the pattern `*.jpg`—Subversion should automatically set the `svn:mime-type` property on those files to `image/jpeg`. Or perhaps any files that match `*.cpp` should have `svn:eol-style` set to `native`, and `svn:keywords` set to `Id`. Automatic property support is perhaps the handiest property-related tool in the Subversion toolbox. See the section called “Config” for more about configuring that support.

File Portability

Fortunately for Subversion users who routinely find themselves on different computers with different operating systems, Subversion's command-line program behaves almost identically on all those systems. If you know how to wield **svn** on one platform, you know how to wield it everywhere.

However, the same is not always true of other general classes of software, or of the actual files you keep in Subversion. For example, on a Windows machine, the definition of a “text file” would be similar to that used on a Linux box, but with a key difference—the character sequences used to mark the ends of the lines of those files. There are other differences, too. Unix platforms have (and Subversion supports) symbolic links; Windows does not. Unix platforms use filesystem permission to determine executability; Windows uses filename extensions.

Because Subversion is in no position to unite the whole world in common definitions and implementations of all of these things, the best it can do is to try to help make your life simpler when you need to work with your versioned files and directories on multiple computers and operating systems. This section describes some of the ways Subversion does this.

File Content Type

Subversion joins the ranks of the many applications which recognize and make use of Multipurpose Internet Mail Extensions (MIME) content types. Besides being a general-purpose storage location for a file's content type, the value of the `svn:mime-type` file property determines some behavioral characteristics of Subversion itself.

Identifying File Types

Various programs on most modern operating systems make assumptions about the type and format of the contents of a file by the file's name, specifically its file extension. For example, files whose names end in `.txt` are generally assumed to be human-readable, able to be understood by simple perusal rather than requiring complex processing to decipher. Files whose names end in `.png`, on the other hand, are assumed to be of the Portable Network Graphics type—not human-readable at all, and sensible only when interpreted by software which understands the PNG format and can render the information in that format as a raster image.

Unfortunately, some of those extensions have changed their meanings over time. When personal computers first appeared, a file named `README.DOC` would have almost certainly been a plaintext file, just like today's `.txt` files. But by the mid-1990's, you could almost bet that a file of that name would not be a plaintext file at all, but instead a Microsoft Word document in a proprietary, non-human-readable format. But this change didn't occur overnight—there was certainly a period of confusion for computer users over what exactly they had in hand when they saw a `.DOC` file. [3]

The popularity of computer networking cast still more doubt on the mapping between a file's name and its content. With information being served across networks and generated dynamically by server-side scripts, there was often no real file per se, and therefore no file name. Web servers, for example, needed some other way to tell browsers what they were downloading so the browser could do something intelligent with that information, whether that was to display the data using a program registered to handle that data type, or to prompt the user for where on the client machine to store the downloaded data.

Eventually, a standard emerged for, among other things, describing the contents of a data stream. In 1996, RFC2045 was published, the first of five RFCs describing MIME. It describes the concept of media types and subtypes, and recommends a syntax for the representation of those types. Today, MIME media types—or “MIME types”—are used almost universally across e-mail applications, Web servers, and other software as the de facto mechanism for clearing up the file content confusion.

For example, one of the benefits that Subversion typically provides is contextual, line-based merging of changes received from the server during an update into your working file. But for files containing non-textual data, there is often no concept of a “line”. So, for versioned files whose `svn:mime-type` property is set to a non-textual MIME type (generally, something that doesn't begin with `text/`, though there are exceptions), Subversion does not attempt to perform contextual merges during updates. Instead, any time you have locally modified a binary working copy file that is also being updated, your file is left untouched and Subversion creates two new files. One file has a `.oldrev` extension and contains the BASE revision of the file. The other file has a `.newrev` extension and contains the contents of the updated revision of the file. This behavior is really for the protection of the user against failed attempts at performing contextual merges on files that simply cannot be contextually merged.

Also, if the `svn:mime-type` property is set, then the Subversion Apache module will use its value to populate the `Content-type:` HTTP header when responding to GET requests. This gives your web browser a crucial clue about how to display a file when you use it to peruse your Subversion repository's contents.

File Executability

[3] You think that was rough? During that same era, WordPerfect also used `.DOC` for their proprietary file format's preferred extension!

On many operating systems, the ability to execute a file as a command is governed by the presence of an execute permission bit. This bit usually defaults to being disabled, and must be explicitly enabled by the user for each file that needs it. But it would be a monumental hassle to have to remember exactly which files in freshly checked-out working copy were supposed to have their executable bits toggled on, and then to have to do that toggling. So, Subversion provides the `svn:executable` property as a way to specify that the executable bit for the file on which that property is set should be enabled, and Subversion honors that request when populating working copies with such files.

This property has no effect on filesystems that have no concept of an executable permission bit, such as FAT32 and NTFS. [4] Also, although it has no defined values, Subversion will force its value to * when setting this property. Finally, this property is valid only on files, not on directories.

End-of-Line Character Sequences

Unless otherwise noted using a versioned file's `svn:mime-type` property, Subversion assumes the file contains human-readable data. Generally speaking, Subversion only uses this knowledge to determine if contextual difference reports for that file are possible. Otherwise, to Subversion, bytes are bytes.

This means that by default, Subversion doesn't pay any attention to the type of *end-of-line (EOL) markers* used in your files. Unfortunately, different operating systems have different conventions about which character sequences represent the end of a line of text in a file. For example, the usual line ending token used by software on the Windows platform is a pair of ASCII control characters—a carriage return (`CR`) followed by a line feed (`LF`). Unix software, however, just uses the `LF` character to denote the end of a line.

Not all of the various tools on these operating systems understand files that contain line endings in a format that differs from the *native line ending style* of the operating system on which they are running. So, typically, Unix programs treat the `CR` character present in Windows files as a regular character (usually rendered as `^M`), and Windows programs combine all of the lines of a Unix file into one giant line because no carriage return-linefeed (or `CRLF`) character combination was found to denote the ends of the lines.

This sensitivity to foreign EOL markers can be frustrating for folks who share a file across different operating systems. For example, consider a source code file, and developers that edit this file on both Windows and Unix systems. If all the developers always use tools which preserve the line ending style of the file, no problems occur.

But in practice, many common tools either fail to properly read a file with foreign EOL markers, or they convert the file's line endings to the native style when the file is saved. If the former is true for a developer, he has to use an external conversion utility (such as **dos2unix** or its companion, **unix2dos**) to prepare the file for editing. The latter case requires no extra preparation. But both cases result in a file that differs from the original quite literally on every line! Prior to committing his changes, the user has two choices. Either he can use a conversion utility to restore the modified file to the same line ending style that it was in before his edits were made. Or, he can simply commit the file—new EOL markers and all.

The result of scenarios like these include wasted time and unnecessary modifications to committed files. Wasted time is painful enough. But when commits change every line in a file, this complicates the job of determining which of those lines were changed in a non-trivial way. Where was that bug really fixed? On what line was a syntax error introduced?

The solution to this problem is the `svn:eol-style` property. When this property is set to a

[4] The Windows filesystems use file extensions (such as `.EXE`, `.BAT`, and `.COM`) to denote executable files.

valid value, Subversion uses it to determine what special processing to perform on the file so that the file's line ending style isn't flip-flopping with every commit that comes from a different operating system. The valid values are:

`native`
: This causes the file to contain the EOL markers that are native to the operating system on which Subversion was run. In other words, if a user on a Windows machine checks out a working copy that contains a file with an `svn:eol-style` property set to `native`, that file will contain `CRLF` EOL markers. A Unix user checking out a working copy which contains the same file will see `LF` EOL markers in his copy of the file.

  Note that Subversion will actually store the file in the repository using normalized `LF` EOL markers regardless of the operating system. This is basically transparent to the user, though.

`CRLF`
: This causes the file to contain `CRLF` sequences for EOL markers, regardless of the operating system in use.

`LF`
: This causes the file to contain `LF` characters for EOL markers, regardless of the operating system in use.

`CR`
: This causes the file to contain `CR` characters for EOL markers, regardless of the operating system in use. This line ending style is not very common. It was used on older Macintosh platforms (on which Subversion doesn't even run).

Ignoring Unversioned Items

In any given working copy there is a good chance that alongside all those versioned files and directories are other files and directories which are neither versioned nor intended to be. Text editors litter directories with backup files. Software compilers generate intermediate—or even final—files which you typically wouldn't bother to version. And users themselves drop various other files and directories wherever they see fit, often in version control working copies.

It's ludicrous to expect Subversion working copies to be somehow impervious to this kind of clutter and impurity. In fact, Subversion counts it as a *feature* that its working copies are just typical directories, just like unversioned trees. But these not-to-be-versioned files and directories can cause some annoyance for Subversion users. For example, because the **svn add** and **svn import** commands act recursively by default, and don't know which files in a given tree you do and don't wish to version, it's easy to accidentally add stuff to version control that you didn't mean to. And because **svn status** reports, by default, every item of interest in a working copy—including unversioned files and directories—its output can get quite noisy where many of these things exist.

So Subversion provides two ways for telling it which files you would prefer that it simply disregard. One of the ways involves the use of Subversion's runtime configuration system (see the section called “Runtime Configuration Area”), and therefore applies to all the Subversion operations which make use of that runtime configuration, generally those performed on a particular computer, or by a particular user of a computer. The other way makes use of Subversion's directory property support, is more tightly bound to the versioned tree itself, and therefore affects everyone who has a working copy of that tree. Both of the mechanisms use file patterns.

The Subversion runtime configuration system provides an option, `global-ignores`, whose

value is a whitespace-delimited collection of file patterns (also known as *globs*). The Subversion client checks these patterns against the names of the files which are candidates for addition to version control, as well as to unversioned files which the **svn status** command notices. If any file's name matches one of the patterns, Subversion will basically act as if the file didn't exist at all. This is really useful for the kinds of files that you almost never want to version, such as editor backup files like Emacs' *~ and .*~ files.

When found on a versioned directory, the `svn:ignore` property is expected to contain a list of newline-delimited file patterns which Subversion should use to determine ignorable objects in that same directory. These patterns do not override those found in the `global-ignores` runtime configuration option, but are instead appended to that list. And it's worth noting again that, unlike the `global-ignores` option, the patterns found in the `svn:ignore` property apply only to the directory on which that property is set, and not to any of its subdirectories. The `svn:ignore` property is a good way to tell Subversion to ignore files that are likely to be present in every user's working copy of that directory, such as compiler output or—to use an example more appropriate to this book—the HTML, PDF, or PostScript files generated as the result of a conversion of some source DocBook XML files to a more legible output format.



Subversion's support for ignorable file patterns extends only to the one-time process of adding unversioned files and directories to version control. Once an object is under Subversion's control, the ignore pattern mechanisms no longer apply to it. In other words, don't expect Subversion to avoid committing changes you've made to a versioned file simply because that file's name matches an ignore pattern—Subversion *always* notices all of its versioned objects.

Ignore Patterns for CVS Users

The Subversion `svn:ignore` property is very similar in syntax and function to the CVS `.cvsignore` file. In fact, if you are migrating a CVS working copy to Subversion, you can directly migrate the ignore patterns by using the `.cvsignore` file as input file to the **svn propset** command:

```
$ svn propset svn:ignore -F .cvsignore .
property 'svn:ignore' set on '.'
$
```

There are, however, some differences in the ways that CVS and Subversion handle ignore patterns. The two systems use the ignore patterns at some different times, and there are slight discrepancies in what the ignore patterns apply to. Also, Subversion does not recognize the use of the `!` pattern as a reset back to having no ignore patterns at all.

The global list of ignore patterns tends to be more a matter of personal taste, and tied more closely to a user's particular tool chain than to the details of any particular working copy's needs. So, the rest of this section will focus on the `svn:ignore` property and its uses.

Say you have the following output from **svn status**:

```
$ svn status calc
 M     calc/button.c
?      calc/calculator
?      calc/data.c
?      calc/debug_log
?      calc/debug_log.1
```

```
?      calc/debug_log.2.gz
?      calc/debug_log.3.gz
```

In this example, you have made some property modifications to `button.c`, but in your working copy you also have some unversioned files: the latest `calculator` program that you've compiled from your source code, a source file named `data.c`, and a set of debugging output log files. Now, you know that your build system always results in the `calculator` program being generated. [5] And you know that your test suite always leaves those debugging log files lying around. These facts are true for all working copies of this project, not just your own. And you know that you aren't interested in seeing those things every time you run **svn status**, and pretty sure that nobody else is interested in them either. So you use **svn propedit svn:ignore calc** to add some ignore patterns to the `calc` directory. For example, you might add this as the new value of the `svn:ignore` property:

```
calculator
debug_log*
```

After you've added this property, you will now have a local property modification on the `calc` directory. But notice what else is different about your **svn status** output:

```
$ svn status
 M     calc
 M     calc/button.c
?      calc/data.c
```

Now, all that cruft is missing from the output! Of course, your `calculator` compiled program and all those logfiles are still in your working copy. Subversion is simply not reminding you that they are present and unversioned. And now with all the uninteresting noise removed from the display, you are left with more interesting items—such as that source code file `data.c` that you probably forgot to add to version control.

Of course, this less-verbose report of your working copy status isn't the only one available. If you actually want to see the ignored files as part of the status report, you can pass the `--no-ignore` option to Subversion:

```
$ svn status --no-ignore
 M     calc
 M     calc/button.c
I      calc/calculator
?      calc/data.c
I      calc/debug_log
I      calc/debug_log.1
I      calc/debug_log.2.gz
I      calc/debug_log.3.gz
```

As mentioned earlier, the list of file patterns to ignore is also used by **svn add** and **svn import**. Both of these operations involve asking Subversion to begin managing some set of files and directories. Rather than force the user to pick and choose which files in a tree she wishes to start versioning, Subversion uses the ignore patterns—both the global and the per-directory lists—to determine which files should not be swept into the version control system as part of a larger recursive addition or import operation. And here again, you can use the `--no-ignore`

[5] Isn't that the whole point of a build system?

option to tell Subversion ignore its ignores list and operate on all the files and directories present.

Keyword Substitution

Subversion has the ability to substitute *keywords*—pieces of useful, dynamic information about a versioned file—into the contents of the file itself. Keywords generally provide information about the last modification made to the file. Because this information changes each time the file changes, and more importantly, just *after* the file changes, it is a hassle for any process except the version control system to keep the data completely up-to-date. Left to human authors, the information would inevitably grow stale.

For example, say you have a document in which you would like to display the last date on which it was modified. You could burden every author of that document to, just before committing their changes, also tweak the part of the document that describes when it was last changed. But sooner or later, someone would forget to do that. Instead, simply ask Subversion to perform keyword substitution on the `LastChangedDate` keyword. You control where the keyword is inserted into your document by placing a *keyword anchor* at the desired location in the file. This anchor is just a string of text formatted as `$KeywordName$`.

All keywords are case-sensitive where they appear as anchors in files: you must use the correct capitalization in order for the keyword to be expanded. You should consider the value of the `svn:keywords` property to be case-sensitive too—certain keyword names will be recognized regardless of case, but this behavior is deprecated.

Subversion defines the list of keywords available for substitution. That list contains the following five keywords, some of which have aliases that you can also use:

`Date`
: This keyword describes the last time the file was known to have been changed in the repository, and is of the form `$Date: 2006-07-22 21:42:37 -0700 (Sat, 22 Jul 2006) $`. It may also be specified as `LastChangedDate`.

`Revision`
: This keyword describes the last known revision in which this file changed in the repository, and looks something like `$Revision: 144 $`. It may also be specified as `LastChangedRevision` or `Rev`.

`Author`
: This keyword describes the last known user to change this file in the repository, and looks something like `$Author: harry $`. It may also be specified as `LastChangedBy`.

`HeadURL`
: This keyword describes the full URL to the latest version of the file in the repository, and looks something like `$HeadURL: http://svn.collab.net/repos/trunk/README $`. It may be abbreviated as `URL`.

`Id`
: This keyword is a compressed combination of the other keywords. Its substitution looks something like `$Id: calc.c 148 2006-07-28 21:30:43Z sally $`, and is interpreted to mean that the file `calc.c` was last changed in revision 148 on the evening of July 28, 2006 by the user `sally`.

Several of the previous descriptions use the phrase “last known” or similar wording. Keep in mind that keyword expansion is a client-side operation, and your client only “knows” about

changes which have occurred in the repository when you update your working copy to include those changes. If you never update your working copy, your keywords will never expand to different values even if those versioned files are being changed regularly in the repository.

Simply adding keyword anchor text to your file does nothing special. Subversion will never attempt to perform textual substitutions on your file contents unless explicitly asked to do so. After all, you might be writing a document [6] about how to use keywords, and you don't want Subversion to substitute your beautiful examples of un-substituted keyword anchors!

To tell Subversion whether or not to substitute keywords on a particular file, we again turn to the property-related subcommands. The `svn:keywords` property, when set on a versioned file, controls which keywords will be substituted on that file. The value is a space-delimited list of the keyword names or aliases found in the previous table.

For example, say you have a versioned file named `weather.txt` that looks like this:

```
Here is the latest report from the front lines.
$LastChangedDate$
$Rev$
Cumulus clouds are appearing more frequently as summer approaches.
```

With no `svn:keywords` property set on that file, Subversion will do nothing special. Now, let's enable substitution of the `LastChangedDate` keyword.

```
$ svn propset svn:keywords "Date Author" weather.txt
property 'svn:keywords' set on 'weather.txt'
$
```

Now you have made a local property modification on the `weather.txt` file. You will see no changes to the file's contents (unless you made some of your own prior to setting the property). Notice that the file contained a keyword anchor for the `Rev` keyword, yet we did not include that keyword in the property value we set. Subversion will happily ignore requests to substitute keywords that are not present in the file, and will not substitute keywords that are not present in the `svn:keywords` property value.

Immediately after you commit this property change, Subversion will update your working file with the new substitute text. Instead of seeing your keyword anchor `$LastChangedDate$`, you'll see its substituted result. That result also contains the name of the keyword, and continues to be bounded by the dollar sign ($) characters. And as we predicted, the `Rev` keyword was not substituted because we didn't ask for it to be.

Note also that we set the `svn:keywords` property to “Date Author” yet the keyword anchor used the alias `$LastChangedDate$` and still expanded correctly.

```
Here is the latest report from the front lines.
$LastChangedDate: 2006-07-22 21:42:37 -0700 (Sat, 22 Jul 2006) $
$Rev$
Cumulus clouds are appearing more frequently as summer approaches.
```

If someone else now commits a change to `weather.txt`, your copy of that file will continue to display the same substituted keyword value as before—until you update your working copy. At that time the keywords in your `weather.txt` file will be re-substituted with information that re-

[6] … or maybe even a section of a book …

flects the most recent known commit to that file.

> **Where's $GlobalRev$?**
>
> New users are often confused by how the `$Rev$` keyword works. Since the repository has a single, globally increasing revision number, many people assume that it is this number which is reflected by the `$Rev$` keyword's value. But `$Rev$` expands to show the last revision in which the file *changed*, not the last revision to which it was updated. Understanding this clears the confusion, but frustration often remains—without the support of a Subversion keyword to do so, how can you automatically get the global revision number into your files?
>
> To do this, you need external processing. Subversion ships with a tool called **svnversion** which was designed for just this purpose. **svnversion** crawls your working copy and generates as output the revision(s) it finds. You can use this program, plus some additional tooling, to embed that revision information into your files. For more information on **svnversion**, see the section called "**svnversion**".

Subversion 1.2 introduced a new variant of the keyword syntax which brought additional, useful—though perhaps atypical—functionality. You can now tell Subversion to maintain a fixed length (in terms of the number of bytes consumed) for the substituted keyword. By using a double-colon (`::`) after the keyword name, followed by a number of space characters, you define that fixed width. When Subversion goes to substitute your keyword for the keyword and its value, it will essentially replace only those space characters, leaving the overall width of the keyword field unchanged. If the substituted value is shorter than the defined field width, there will be extra padding characters (spaces) at the end of the substituted field; if it is too long, it is truncated with a special hash (#) character just before the final dollar sign terminator.

For example, say you have a document in which you have some section of tabular data reflecting the document's Subversion keywords. Using the original Subversion keyword substitution syntax, your file might look something like:

```
$Rev$:     Revision of last commit
$Author$:  Author of last commit
$Date$:    Date of last commit
```

Now, that looks nice and tabular at the start of things. But when you then commit that file (with keyword substitution enabled, of course), you see:

```
$Rev: 12 $:     Revision of last commit
$Author: harry $:  Author of last commit
$Date: 2006-03-15 02:33:03 -0500 (Wed, 15 Mar 2006) $:    Date of last commit
```

The result is not so beautiful. And you might be tempted to then adjust the file after the substitution so that it again looks tabular. But that only holds as long as the keyword values are the same width. If the last committed revision rolls into a new place value (say, from 99 to 100), or if another person with a longer username commits the file, stuff gets all crooked again. However, if you are using Subversion 1.2 or better, you can use the new fixed-length keyword syntax, define some field widths that seem sane, and now your file might look like this:

```
$Rev::               $:  Revision of last commit
$Author::            $:  Author of last commit
$Date::              $:  Date of last commit
```

You commit this change to your file. This time, Subversion notices the new fixed-length keyword syntax, and maintains the width of the fields as defined by the padding you placed between the double-colon and the trailing dollar sign. After substitution, the width of the fields is completely unchanged—the short values for `Rev` and `Author` are padded with spaces, and the long `Date` field is truncated by a hash character:

```
$Rev:: 13            $:  Revision of last commit
$Author:: harry      $:  Author of last commit
$Date:: 2006-03-15 0#$:  Date of last commit
```

The use of fixed-length keywords is especially handy when performing substitutions into complex file formats that themselves use fixed-length fields for data, or for which the stored size of a given data field is overbearingly difficult to modify from outside the format's native application (such as for Microsoft Office documents).



Be aware that because the width of a keyword field is measured in bytes, the potential for corruption of multi-byte values exists. For example, a username which contains some multi-byte UTF-8 characters might suffer truncation in the middle of the string of bytes which make up one of those characters. The result will be a mere truncation when viewed at the byte level, but will likely appear as a string with an incorrect or garbled final character when viewed as UTF-8 text. It is conceivable that certain applications, when asked to load the file, would notice the broken UTF-8 text and deem the entire file corrupt, refusing to operate on the file altogether. So, when limiting keywords to a fixed size, choose a size that allows for this type of byte-wise expansion.

Locking

Subversion's copy-modify-merge version control model lives and dies on its data merging algorithms, specifically on how well those algorithms perform when trying to resolve conflicts caused by multiple users modifying the same file concurrently. Subversion itself provides only one such algorithm, a three-way differencing algorithm which is smart enough to handle data at a granularity of a single line of text. Subversion also allows you to supplement its content merge processing with external differencing utilities (as described in the section called “External diff3”), some of which may do an even better job, perhaps providing granularity of a word or a single character of text. But common among those algorithms is that they generally work only on text files. The landscape starts to look pretty grim when you start talking about content merges of non-textual file formats. And when you can't find a tool that can handle that type of merging, you begin to run into problems with the copy-modify-merge model.

Let's look at a real-life example of where this model runs aground. Harry and Sally are both graphic designers working on the same project, a bit of marketing collateral for an automobile mechanic. Central to the design of a particular poster is an image of a car in need of some body work, stored in a file using the PNG image format. The poster's layout is almost finished, and both Harry and Sally are pleased with the particular photo they chose for their damaged car—a baby blue 1967 Ford Mustang with an unfortunate bit of crumpling on the left front fender.

Now, as is common in graphic design work, there's a change in plans which causes the car's color to be a concern. So Sally updates her working copy to `HEAD`, fires up her photo editing software, and sets about tweaking the image so that the car is now cherry red. Meanwhile,

Harry, feeling particularly inspired that day, decides that the image would have greater impact if the car also appears to have suffered greater impact. He, too, updates to HEAD, and then draws some cracks on the vehicle's windshield. He manages to finish his work before Sally finishes hers, and after admiring the fruits of his undeniable talent, commits the modified image. Shortly thereafter, Sally is finished with the car's new finish, and tries to commit her changes. But, as expected, Subversion fails the commit, informing Sally that now her version of the image is out of date.

Here's where the difficulty sets in. Were Harry and Sally making changes to a text file, Sally would simply update her working copy, receiving Harry's changes in the process. In the worst possible case, they would have modified the same region of the file, and Sally would have to work out by hand the proper resolution to the conflict. But these aren't text files—they are binary images. And while it's a simple matter to describe what one would expect the results of this content merge to be, there is precious little chance that any software exists which is smart enough to examine the common baseline image that each of these graphic artists worked against, the changes that Harry made, and the changes that Sally made, and spit out an image of a busted-up red Mustang with a cracked windshield!

Clearly, things would have gone more smoothly if Harry and Sally had serialized their modifications to the image—if, say, Harry had waited to draw his windshield cracks on Sally's now-red car, or if Sally had tweaked the color of a car whose windshield was already cracked. As is discussed in the section called “The Copy-Modify-Merge Solution”, most of these types of problems go away entirely where perfect communication between Harry and Sally exists. [7] But as one's version control system is, in fact, one form of communication, it follows that having that software facilitate the serialization of non-parallelizable editing efforts is no bad thing. This where Subversion's implementation of the lock-modify-unlock model steps into the spotlight. This is where we talk about Subversion's *locking* feature, which is similar to the “reserved checkouts” mechanisms of other version control systems.

Subversion's locking feature serves two main purposes:

- *Serializing access to a versioned object*. By allowing a user to programmatically claim the exclusive right to change to a file in the repository, that user can be reasonably confident that energy invested on unmergeable changes won't be wasted—his commit of those changes will succeed.

- *Aiding communication*. By alerting other users that serialization is in effect for a particular versioned object, those other users can reasonably expect that the object is about to be changed by someone else, and they, too, can avoid wasting their time and energy on unmergeable changes that won't be committable due to eventual out-of-dateness.

When referring to Subversion's locking feature, one is actually talking about a fairly diverse collection of behaviors which include the ability to lock a versioned file [8] (claiming the exclusive right to modify the file), to unlock that file (yielding that exclusive right to modify), to see reports about which files are locked and by whom, to annotate files for which locking before editing is strongly advised, and so on. In this section, we'll cover all of these facets of the larger locking feature.

The three meanings of “lock”

In this section, and almost everywhere in this book, the words “lock” and “locking” describe a mechanism for mutual exclusion between users to avoid clashing commits. Un-

[7] Communication wouldn't have been such bad medicine for Harry and Sally's Hollywood namesakes, either, for that matter.
[8] Subversion does not currently allow locks on directories.

fortunately, there are two other sorts of “lock” with which Subversion, and therefore this book, sometimes needs to be concerned.

The first is *working copy locks*, used internally by Subversion to prevent clashes between multiple Subversion clients operating on the same working copy. This is the sort of lock indicated by an L in the third column of **svn status** output, and removed by the **svn cleanup** command, as described in the section called “Sometimes You Just Need to Clean Up”.

Secondly, there are *database locks*, used internally by the Berkeley DB backend to prevent clashes between multiple programs trying to access the database. This is the sort of lock whose unwanted persistence after an error can cause a repository to be “wedged”, as described in the section called “Berkeley DB Recovery”.

You can generally forget about these other kinds of locks until something goes wrong that requires you to care about them. In this book, “lock” means the first sort unless the contrary is either clear from context or explicitly stated.

Creating locks

In the Subversion repository, a *lock* is a piece of metadata which grants exclusive access to one user to change a file. This user is said to be the *lock owner*. Each lock also has a unique identifier, typically a long string of characters, known as the *lock token*. The repository manages locks, ultimately handling their creation, enforcement, and removal. If any commit transaction attempts to modify or delete a locked file (or delete one of the parent directories of the file), the repository will demand two pieces of information—that the client performing the commit be authenticated as the lock owner, and that the lock token has been provided as part of the commit process as a sort of proof that client knows which lock it is using.

To demonstrate lock creation, let's refer back to our example of multiple graphic designers working with on the same binary image files. Harry has decided to change a JPEG image. To prevent other people from committing changes to the file while he is modifying it (as well as alerting them that he is about to change it), he locks the file in the repository using the **svn lock** command.

```
$ svn lock banana.jpg -m "Editing file for tomorrow's release."
'banana.jpg' locked by user 'harry'.
$
```

There are a number of new things demonstrated in the previous example. First, notice that Harry passed the `--message (-m)` option to **svn lock**. Similar to **svn commit**, the **svn lock** command can take comments (either via `--message (-m)` or `--file (-F)`) to describe the reason for locking the file. Unlike **svn commit**, however, **svn lock** will not demand a message by launching your preferred text editor. Lock comments are optional, but still recommended to aid communication.

Secondly, the lock attempt succeeded. This means that the file wasn't already locked, and that Harry had the latest version of the file. If Harry's working copy of the file had been out-of-date, the repository would have rejected the request, forcing Harry to **svn update** and reattempt the locking command. The locking command would also have failed if the file already been locked by someone else.

As you can see, the **svn lock** command prints confirmation of the successful lock. At this

point, the fact that the file is locked becomes apparent in the output of the **svn status** and **svn info** reporting subcommands.

```
$ svn status
     K banana.jpg

$ svn info banana.jpg
Path: banana.jpg
Name: banana.jpg
URL: http://svn.example.com/repos/project/banana.jpg
Repository UUID: edb2f264-5ef2-0310-a47a-87b0ce17a8ec
Revision: 2198
Node Kind: file
Schedule: normal
Last Changed Author: frank
Last Changed Rev: 1950
Last Changed Date: 2006-03-15 12:43:04 -0600 (Wed, 15 Mar 2006)
Text Last Updated: 2006-06-08 19:23:07 -0500 (Thu, 08 Jun 2006)
Properties Last Updated: 2006-06-08 19:23:07 -0500 (Thu, 08 Jun 2006)
Checksum: 3b110d3b10638f5d1f4fe0f436a5a2a5
Lock Token: opaquelocktoken:0c0f600b-88f9-0310-9e48-355b44d4a58e
Lock Owner: harry
Lock Created: 2006-06-14 17:20:31 -0500 (Wed, 14 Jun 2006)
Lock Comment (1 line):
Editing file for tomorrow's release.

$
```

That the **svn info** command, which does not contact the repository when run against working copy paths, can display the lock token reveals an important fact about lock tokens—that they are cached in the working copy. The presence of the lock token is critical. It gives the working copy authorization to make use of the lock later on. Also, the **svn status** command shows a K next to the file (short for locKed), indicating that the lock token is present.

> **Regarding lock tokens**
>
> A lock token isn't an authentication token, so much as an *authorization* token. The token isn't a protected secret. In fact, a lock's unique token is discoverable by anyone who runs **svn info URL**. A lock token is special only when it lives inside a working copy. It's proof that the lock was created in that particular working copy, and not somewhere else by some other client. Merely authenticating as the lock owner isn't enough to prevent accidents.
>
> For example, suppose you lock a file using a computer at your office, but leave work for the day before you finish your changes to that file. It should not be possible to accidentally commit changes to that same file from your home computer later that evening simply because you've authenticated as the lock's owner. In other words, the lock token prevents one piece of Subversion-related software from undermining the work of another. (In our example, if you really need to change the file from an alternate working copy, you would need to *break* the lock and re-lock the file.)

Now that Harry has locked `banana.jpg`, Sally is unable to change or delete that file:

```
$ svn delete banana.jpg
D         banana.jpg
$ svn commit -m "Delete useless file."
```

```
Deleting       banana.jpg
svn: Commit failed (details follow):
svn: DELETE of
'/repos/project/!svn/wrk/64bad3a9-96f9-0310-818a-df4224ddc35d/banana.jpg':
423 Locked (http://svn.example.com)
$
```

But Harry, after touching up the banana's shade of yellow, is able to commit his changes to the file. That's because he authenticates as the lock owner, and also because his working copy holds the correct lock token:

```
$ svn status
M    K banana.jpg
$ svn commit -m "Make banana more yellow"
Sending        banana.jpg
Transmitting file data .
Committed revision 2201.
$ svn status
$
```

Notice that after the commit is finished, **svn status** shows that the lock token is no longer present in working copy. This is the standard behavior of **svn commit**—it searches the working copy (or list of targets, if you provide such a list) for local modifications, and sends all the lock tokens it encounters during this walk to the server as part of the commit transaction. After the commit completes successfully, all of the repository locks that were mentioned are released—*even on files that weren't committed*. This is meant to discourage users from being sloppy about locking, or from holding locks for too long. If Harry haphazardly locks thirty files in a directory named `images` because he's unsure of which files he needs to change, yet only only changes four of those files, when he runs **svn commit images**, the process will still release all thirty locks.

This behavior of automatically releasing locks can be overridden with the `--no-unlock` option to **svn commit**. This is best used for those times when you want to commit changes, but still plan to make more changes and thus need to retain existing locks. You can also make this your default behavior by setting the `no-unlock` runtime configuration option (see the section called “Runtime Configuration Area”).

Of course, locking a file doesn't oblige one to commit a change to it. The lock can be released at any time with a simple **svn unlock** command:

```
$ svn unlock banana.c
'banana.c' unlocked.
```

Discovering locks

When a commit fails due to someone else's locks, it's fairly easy to learn about them. The easiest of these is **svn status --show-updates**:

```
$ svn status -u
M             23   bar.c
M    O        32   raisin.jpg
       *      72   foo.h
Status against revision:     105
$
```

In this example, Sally can see not only that her copy of `foo.h` is out-of-date, but that one of the two modified files she plans to commit is locked in the repository. The `O` symbol stands for "Other", meaning that a lock exists on the file, and was created by somebody else. If she were to attempt a commit, the lock on `raisin.jpg` would prevent it. Sally is left wondering who made the lock, when, and why. Once again, **svn info** has the answers:

```
$ svn info http://svn.example.com/repos/project/raisin.jpg
Path: raisin.jpg
Name: raisin.jpg
URL: http://svn.example.com/repos/project/raisin.jpg
Repository UUID: edb2f264-5ef2-0310-a47a-87b0ce17a8ec
Revision: 105
Node Kind: file
Last Changed Author: sally
Last Changed Rev: 32
Last Changed Date: 2006-01-25 12:43:04 -0600 (Sun, 25 Jan 2006)
Lock Token: opaquelocktoken:fc2b4dee-98f9-0310-abf3-653ff3226e6b
Lock Owner: harry
Lock Created: 2006-02-16 13:29:18 -0500 (Thu, 16 Feb 2006)
Lock Comment (1 line):
Need to make a quick tweak to this image.
$
```

Just as **svn info** can be used to examine objects in the working copy, it can also be used to examine objects in the repository. If the main argument to **svn info** is a working copy path, then all of the working copy's cached information is displayed; any mention of a lock means that the working copy is holding a lock token (if a file is locked by another user or in another working copy, **svn info** on a working copy path will show no lock information at all). If the main argument to **svn info** is a URL, then the information reflects the latest version of an object in the repository, and any mention of a lock describes the current lock on the object.

So in this particular example, Sally can see that Harry locked the file on February 16th to "make a quick tweak". It being June, she suspects that he probably forgot all about the lock. She might phone Harry to complain and ask him to release the lock. If he's unavailable, she might try to forcibly break the lock herself or ask an administrator to do so.

Breaking and stealing locks

A repository lock isn't sacred—in Subversion's default configuration state, locks can be released not only by the person who created them, but by anyone at all. When somebody other than the original lock creator destroys a lock, we refer to this as *breaking* the lock.

From the administrator's chair, it's simple to break locks. The **svnlook** and **svnadmin** programs have the ability to display and remove locks directly from the repository. (For more information about these tools, see the section called "An Administrator's Toolkit".)

```
$ svnadmin lslocks /usr/local/svn/repos
Path: /project2/images/banana.jpg
UUID Token: opaquelocktoken:c32b4d88-e8fb-2310-abb3-153ff1236923
Owner: frank
Created: 2006-06-15 13:29:18 -0500 (Thu, 15 Jun 2006)
Expires:
Comment (1 line):
Still improving the yellow color.
```

```
Path: /project/raisin.jpg
UUID Token: opaquelocktoken:fc2b4dee-98f9-0310-abf3-653ff3226e6b
Owner: harry
Created: 2006-02-16 13:29:18 -0500 (Thu, 16 Feb 2006)
Expires:
Comment (1 line):
Need to make a quick tweak to this image.

$ svnadmin rmlocks /usr/local/svn/repos /project/raisin.jpg
Removed lock on '/project/raisin.jpg'.
$
```

The more interesting option is allowing users to break each other's locks over the network. To do this, Sally simply needs to pass the `--force` to the unlock command:

```
$ svn status -u
M             23   bar.c
M    O        32   raisin.jpg
       *      72   foo.h
Status against revision:     105
$ svn unlock raisin.jpg
svn: 'raisin.jpg' is not locked in this working copy
$ svn info raisin.jpg | grep URL
URL: http://svn.example.com/repos/project/raisin.jpg
$ svn unlock http://svn.example.com/repos/project/raisin.jpg
svn: Unlock request failed: 403 Forbidden (http://svn.example.com)
$ svn unlock --force http://svn.example.com/repos/project/raisin.jpg
'raisin.jpg' unlocked.
$
```

Now, Sally's initial attempt to unlock failed because she ran **svn unlock** directly on her working copy of the file, and no lock token was present. To remove the lock directly from the repository, she needs to pass a URL to **svn unlock**. Her first attempt to unlock the URL fails, because she can't authenticate as the lock owner (nor does she have the lock token). But when she passes `--force`, the authentication and authorization requirements are ignored, and the remote lock is broken.

Simply breaking a lock may not be enough. In the running example, Sally may not only want to break Harry's long-forgotten lock, but re-lock the file for her own use. She can accomplish this by running **svn unlock --force** and then **svn lock** back-to-back, but there's a small chance that somebody else might lock the file between the two commands. The simpler thing to is *steal* the lock, which involves breaking and re-locking the file all in one atomic step. To do this, Sally passes the `--force` option to **svn lock**:

```
$ svn lock raisin.jpg
svn: Lock request failed: 423 Locked (http://svn.example.com)
$ svn lock --force raisin.jpg
'raisin.jpg' locked by user 'sally'.
$
```

In any case, whether the lock is broken or stolen, Harry may be in for a surprise. Harry's working copy still contains the original lock token, but that lock no longer exists. The lock token is said to be *defunct*. The lock represented by the lock token has either been broken (no longer in the repository), or stolen (replaced with a different lock). Either way, Harry can see this by asking **svn status** to contact the repository:

```
$ svn status
     K raisin.jpg
$ svn status -u
     B        32   raisin.jpg
$ svn update
  B  raisin.jpg
$ svn status
$
```

If the repository lock was broken, then **svn status --show-updates** displays a B (Broken) symbol next to the file. If a new lock exists in place of the old one, then a T (sTolen) symbol is shown. Finally, **svn update** notices any defunct lock tokens and removes them from the working copy.

Locking Policies

Different systems have different notions of how strict a lock should be. Some folks argue that locks must be strictly enforced at all costs, releasable only by the original creator or administrator. They argue that if anyone can break a lock, then chaos runs rampant and the whole point of locking is defeated. The other side argues that locks are first and foremost a communication tool. If users are constantly breaking each others' locks, then it represents a cultural failure within the team and the problem falls outside the scope of software enforcement.

Subversion defaults to the “softer” approach, but still allows administrators to create stricter enforcement policies through the use of hook scripts. In particular, the `pre-lock` and `pre-unlock` hooks allow administrators to decide when lock creation and lock releases are allowed to happen. Depending on whether or not a lock already exists, these two hooks can decide whether or not to allow a certain user to break or steal a lock. The `post-lock` and `post-unlock` hooks are also available, and can be used to send email after locking actions. To learn more about repository hooks, see the section called “Implementing Repository Hooks”.

Lock Communication

We've seen how **svn lock** and **svn unlock** can be used to create, release, break, and steal locks. This satisfies the goal of serializing commit access to a file. But what about the larger problem of preventing wasted time?

For example, suppose Harry locks an image file and then begins editing it. Meanwhile, miles away, Sally wants to do the same thing. She doesn't think to run **svn status --show-updates**, so she has no idea that Harry has already locked the file. She spends hours editing the file, and when she tries to commit her change, she discovers that either the file is locked or that she's out-of-date. Regardless, her changes aren't mergeable with Harry's. One of these two people has to throw away their work, and a lot of time has been wasted.

Subversion's solution to this problem is to provide a mechanism to remind users that a file ought to be locked *before* the editing begins. The mechanism is a special property, `svn:needs-lock`. If that property is attached to a file (regardless of its value, which is irrelevant), then Subversion will try to use filesystem-level permissions to make the file read-only—unless, of course, the user has explicitly locked the file. When a lock token is present (as a result of running **svn lock**), the file becomes read-write. When the lock is released, the file becomes read-only again.

The theory, then, is that if the image file has this property attached, then Sally would immediately notice something is strange when she opens the file for editing: many applications alert users immediately when a read-only file is opened for editing, and nearly all would prevent her from saving changes to the file. This reminds her to lock the file before editing, whereby she discovers the pre-existing lock:

```
$ /usr/local/bin/gimp raisin.jpg
gimp: error: file is read-only!
$ ls -l raisin.jpg
-r--r--r--   1 sally   sally   215589 Jun  8 19:23 raisin.jpg
$ svn lock raisin.jpg
svn: Lock request failed: 423 Locked (http://svn.example.com)
$ svn info http://svn.example.com/repos/project/raisin.jpg | grep Lock
Lock Token: opaquelocktoken:fc2b4dee-98f9-0310-abf3-653ff3226e6b
Lock Owner: harry
Lock Created: 2006-06-08 07:29:18 -0500 (Thu, 08 June 2006)
Lock Comment (1 line):
Making some tweaks.  Locking for the next two hours.
$
```



Users and administrators alike are encouraged to attach the `svn:needs-lock` property to any file which cannot be contextually merged. This is the primary technique for encouraging good locking habits and preventing wasted effort.

Note that this property is a communication tool which works independently from the locking system. In other words, any file can be locked, whether or not this property is present. And conversely, the presence of this property doesn't make the repository require a lock when committing.

Unfortunately, the system isn't flawless. It's possible that even when a file has the property, the read-only reminder won't always work. Sometimes applications misbehave and “hijack” the read-only file, silently allowing users to edit and save the file anyway. There's not much that Subversion can do in this situation—at the end of the day, there's simply no substitution for good interpersonal communication. [9]

Externals Definitions

Sometimes it is useful to construct a working copy that is made out of a number of different checkouts. For example, you may want different subdirectories to come from different locations in a repository, or perhaps from different repositories altogether. You could certainly set up such a scenario by hand—using **svn checkout** to create the sort of nested working copy structure you are trying to achieve. But if this layout is important for everyone who uses your repository, every other user will need to perform the same checkout operations that you did.

Fortunately, Subversion provides support for *externals definitions*. An externals definition is a mapping of a local directory to the URL—and ideally a particular revision—of a versioned directory. In Subversion, you declare externals definitions in groups using the `svn:externals` property. You can create or modify this property using **svn propset** or **svn propedit** (see the section called “Manipulating Properties”). It can be set on any versioned directory, and its value is a multi-line table of subdirectories (relative to the versioned directory on which the

[9] Except, perhaps, a classic Vulcan mind-meld.

property is set), optional revision flags, and fully qualified, absolute Subversion repository URLs.

```
$ svn propget svn:externals calc
third-party/sounds             http://sounds.red-bean.com/repos
third-party/skins              http://skins.red-bean.com/repositories/skinproj
third-party/skins/toolkit -r21 http://svn.red-bean.com/repos/skin-maker
```

The convenience of the `svn:externals` property is that once it is set on a versioned directory, everyone who checks out a working copy with that directory also gets the benefit of the externals definition. In other words, once one person has made the effort to define those nested working copy checkouts, no one else has to bother—Subversion will, upon checkout of the original working copy, also check out the external working copies.



The relative target subdirectories of externals definitions *must not* already exist on your or other users' systems—Subversion will create them when it checks out the external working copy.

Note the previous externals definition example. When someone checks out a working copy of the `calc` directory, Subversion also continues to check out the items found in its externals definition.

```
$ svn checkout http://svn.example.com/repos/calc
A  calc
A  calc/Makefile
A  calc/integer.c
A  calc/button.c
Checked out revision 148.

Fetching external item into calc/third-party/sounds
A  calc/third-party/sounds/ding.ogg
A  calc/third-party/sounds/dong.ogg
A  calc/third-party/sounds/clang.ogg
…
A  calc/third-party/sounds/bang.ogg
A  calc/third-party/sounds/twang.ogg
Checked out revision 14.

Fetching external item into calc/third-party/skins
…
```

If you need to change the externals definition, you can do so using the regular property modification subcommands. When you commit a change to the `svn:externals` property, Subversion will synchronize the checked-out items against the changed externals definition when you next run **svn update**. The same thing will happen when others update their working copies and receive your changes to the externals definition.



Because the `svn:externals` property has a multiline value, we strongly recommend that you use **svn propedit** instead of **svn propset**.



You should seriously consider using explicit revision numbers in all of your externals definitions. Doing so means that you get to decide when to pull down a different snapshot of external information, and exactly which snapshot to pull. Besides

avoiding the surprise of getting changes to third-party repositories that you might not have any control over, using explicit revision numbers also means that as you backdate your working copy to a previous revision, your externals definitions will also revert to the way they looked in that previous revision, which in turn means that the external working copies will be updated to match they way *they* looked back when your repository was at that previous revision. For software projects, this could be the difference between a successful and a failed build of an older snapshot of your complex codebase.

The **svn status** command also recognizes externals definitions, displaying a status code of `X` for the disjoint subdirectories into which externals are checked out, and then recursing into those subdirectories to display the status of the external items themselves.

The support that exists for externals definitions in Subversion is less than ideal, though. First, an externals definition can only point to directories, not files. Second, the externals definition cannot point to relative paths (paths like `../../skins/myskin`). Third, the working copies created via the externals definition support are still disconnected from the primary working copy (on whose versioned directories the `svn:externals` property was actually set). And Subversion still only truly operates on non-disjoint working copies. So, for example, if you want to commit changes that you've made in one or more of those external working copies, you must run **svn commit** explicitly on those working copies—committing on the primary working copy will not recurse into any external ones.

Also, since the definitions themselves use absolute URLs, moving or copying a directory to which they are attached will not affect what gets checked out as an external (though the relative local target subdirectory will, of course, move with renamed directory). This can be confusing—even frustrating—in certain situations. For example, say you have a top-level directory named `my-project`, and you've created an externals definition on one of its subdirectories (`my-project/some-dir`) which tracks the latest revision of another of its subdirectories (`my-project/external-dir`).

```
$ svn checkout http://svn.example.com/projects .
A    my-project
A    my-project/some-dir
A    my-project/external-dir
…
Fetching external item into 'my-project/some-dir/subdir'
Checked out external at revision 11.

Checked out revision 11.
$ svn propget svn:externals my-project/some-dir
subdir http://svn.example.com/projects/my-project/external-dir

$
```

Now you use **svn move** to rename the `my-project` directory. At this point, your externals definition will still refer to a path under the `my-project` directory, even though that directory no longer exists.

```
$ svn move -q my-project renamed-project
$ svn commit -m "Rename my-project to renamed-project."
Deleting       my-project
Adding         my-renamed-project

Committed revision 12.
```

```
$ svn update

Fetching external item into 'renamed-project/some-dir/subdir'
svn: Target path does not exist
$
```

Also, the absolute URLs that externals definitions use can cause problems with repositories that are available via multiple URL schemes. For example, if your Subversion server is configured to allow everyone to check out the repository over `http://` or `https://`, but only allow commits to come in via `https://`, you have an interesting problem on your hands. If your externals definitions use the `http://` form of the repository URLs, you won't be able to commit anything from the working copies created by those externals. On the other hand, if they use the `https://` form of the URLs, anyone who might be checking out via `http://` because their client doesn't support `https://` will be unable to fetch the external items. Be aware, too, that if you need to re-parent your working copy (using **svn switch --relocate**), externals definitions will *not* also be re-parented.

Finally, there might be times when you would prefer that **svn** subcommands would not recognize, or otherwise operate upon, the external working copies. In those instances, you can pass the `--ignore-externals` option to the subcommand.

Peg and Operative Revisions

We copy, move, rename, and completely replace files and directories on our computers all the time. And your version control system shouldn't get in the way of your doing these things with your version-controlled files and directories, either. Subversion's file management support is quite liberating, affording almost as much flexibility for versioned files as you'd expect when manipulating your unversioned ones. But that flexibility means that across the lifetime of your repository, a given versioned object might have many paths, and a given path might represent several entirely different versioned objects. And this introduces a certain level of complexity to your interactions with those paths and objects.

Subversion is pretty smart about noticing when an object's version history includes such “changes of address”. For example, if you ask for the revision history log of a particular file that was renamed last week, Subversion happily provides all those logs—the revision in which the rename itself happened, plus the logs of relevant revisions both before and after that rename. So, most of the time, you don't even have to think about such things. But occasionally, Subversion needs your help to clear up ambiguities.

The simplest example of this occurs when a directory or file is deleted from version control, and then a new directory or file is created with the same name and added to version control. Clearly the thing you deleted and the thing you later added aren't the same thing. They merely happen to have had the same path, `/trunk/object` for example. What, then, does it mean to ask Subversion about the history of `/trunk/object`? Are you asking about the thing currently at that location, or the old thing you deleted from that location? Are you asking about the operations that have happened to *all* the objects that have ever lived at that path? Clearly, Subversion needs a hint about what you really want.

And thanks to moves, versioned object history can get far more twisted than that, even. For example, you might have a directory named `concept`, containing some nascent software project you've been toying with. Eventually, though, that project matures to the point that the idea seems to actually have some wings, so you do the unthinkable and decide to give the project a name. [10] Let's say you called your software Frabnaggilywort. At this point, it makes sense to rename the directory to reflect the project's new name, so `concept` is renamed to `frabnag-`

[10] “You're not supposed to name it. Once you name it, you start getting attached to it.”—Mike Wazowski

`gilywort`. Life goes on, Frabnaggilywort releases a 1.0 version, and is downloaded and used daily by hordes of people aiming to improve their lives.

It's a nice story, really, but it doesn't end there. Entrepreneur that you are, you've already got another think in the tank. So you make a new directory, `concept`, and the cycle begins again. In fact, the cycle begins again many times over the years, each time starting with that old `concept` directory, then sometimes seeing that directory renamed as the idea cures, sometimes seeing it deleted when you scrap the idea. Or, to get really sick, maybe you rename `concept` to something else for a while, but later rename the thing back to `concept` for some reason.

In scenarios like these, attempting to instruct Subversion to work with these re-used paths can be a little like instructing a motorist in Chicago's West Suburbs to drive east down Roosevelt Road and turn left onto Main Street. In a mere twenty minutes, you can cross “Main Street” in Wheaton, Glen Ellyn, and Lombard. And no, they aren't the same street. Our motorist—and our Subversion—need a little more detail in order to do the right thing.

In version 1.1, Subversion introduced a way for you to tell it exactly which Main Street you meant. It's called the *peg revision*, and it is a revision provided to Subversion for the sole purpose of identifying a unique line of history. Because at most one versioned object may occupy a path at any given time—or, more precisely, in any one revision—the combination of a path and a peg revision is all that is needed to refer to a specific line of history. Peg revisions are specified to the Subversion command-line client using *at syntax*, so called because the syntax involves appending an “at sign” (@) and the peg revision to the end of the path with which the revision is associated.

But what of the `--revision (-r)` of which we've spoken so much in this book? That revision (or set of revisions) is called the *operative revision* (or *operative revision range*). Once a particular line of history has been identified using a path and peg revision, Subversion performs the requested operation using the operative revision(s). To map this to our Chicagoland streets analogy, if we are told to go to 606 N. Main Street in Wheaton, [11] we can think of “Main Street” as our path and “Wheaton” as our peg revision. These two pieces of information identify a unique path which can travelled (north or south on Main Street), and will keep us from travelling up and down the wrong Main Street in search of our destination. Now we throw in “606 N.” as our operative revision, of sorts, and we know *exactly* where to go.

The peg revision algorithm

The Subversion command-line performs the peg revision algorithm any time it needs to resolve possible ambiguities in the paths and revisions provided to it. Here's an example of such an invocation:

```
$ svn command -r OPERATIVE-REV item@PEG-REV
```

If *`OPERATIVE-REV`* is older than *`PEG-REV`*, then the algorithm is as follows:

- Locate *`item`* in the revision identified by *`PEG-REV`*. There can be only one such object.
- Trace the object's history backwards (through any possible renames) to its ancestor in the revision *`OPERATIVE-REV`*.
- Perform the requested action on that ancestor, wherever it is located, or whatever its

[11] 606 N. Main Street, Wheaton, Illinois, is the home of the Wheaton History Center. Get it—“History Center”? It seemed appropriate….

name might be or have been at that time.

But what if *OPERATIVE-REV* is *younger* than *PEG-REV*? Well, that adds some complexity to the theoretical problem of locating the path in *OPERATIVE-REV*, because the path's history could have forked multiple times (thanks to copy operations) between *PEG-REV* and *OPERATIVE-REV*. And that's not all—Subversion doesn't store enough information to performantly trace an object's history forward, anyway. So the algorithm is a little different:

- Locate *item* in the revision identified by *OPERATIVE-REV*. There can be only one such object.
- Trace the object's history backwards (through any possible renames) to its ancestor in the revision *PEG-REV*.
- Verify that the object's location (path-wise) in *PEG-REV* is the same as it is in *OPERATIVE-REV*. If that's the case, then at least the two locations are known to be directly related, so perform the requested action on the location in *OPERATIVE-REV*. Otherwise, relatedness was not established, so error out with a loud complaint that no viable location was found. (Someday, we expect that Subversion will be able to handle this usage scenario with more flexibility and grace.)

Note that even when you don't explicitly supply a peg revision or operative revision, they are still present. For your convenience, the default peg revision is BASE for working copy items and HEAD for repository URLs. And when no operative revision is provided, it defaults to being the same revision as the peg revision.

Say that long ago we created our repository, and in revision 1 added our first `concept` directory, plus an `IDEA` file in that directory talking about the concept. After several revisions in which real code was added and tweaked, we, in revision 20, renamed this directory to `frabnaggilywort`. By revision 27, we had a new concept, a new `concept` directory to hold it, and a new `IDEA` file to describe it. And then five years and twenty thousand revisions flew by, just like they would in any good romance story.

Now, years later, we wonder what the `IDEA` file looked like back in revision 1. But Subversion needs to know if we are asking about how the *current* file looked back in revision 1, or if we are asking for the contents of whatever file lived at `concepts/IDEA` in revision 1. Certainly those questions have different answers, and because of peg revisions, you can ask either of them. To find out how the current `IDEA` file looked in that old revision, you run:

```
$ svn cat -r 1 concept/IDEA
svn: Unable to find repository location for 'concept/IDEA' in revision 1
```

Of course, in this example, the current `IDEA` file didn't exist yet in revision 1, so Subversion gives an error. The command above is shorthand for a longer notation which explicitly lists a peg revision. The expanded notation is:

```
$ svn cat -r 1 concept/IDEA@BASE
svn: Unable to find repository location for 'concept/IDEA' in revision 1
```

And when executed, it has the expected results.

The perceptive reader is probably wondering at this point if the peg revision syntax causes problems for working copy paths or URLs that actually have at signs in them. After all, how does **svn** know whether `news@11` is the name of a directory in my tree, or just a syntax for "revision 11 of `news`"? Thankfully, while **svn** will always assume the latter, there is a trivial workaround. You need only append an at sign to the end of the path, such as `news@11@`. **svn** only cares about the last at sign in the argument, and it is not considered illegal to omit a literal peg revision specifier after that at sign. This workaround even applies to paths that end in an at sign—you would use `filename@@` to talk about a file named `filename@`.

Let's ask the other question, then—in revision 1, what were the contents of whatever file occupied the address `concepts/IDEA` at the time? We'll use an explicit peg revision to help us out.

```
$ svn cat concept/IDEA@1
The idea behind this project is to come up with a piece of software
that can frab a naggily wort.  Frabbing naggily worts is tricky
business, and doing it incorrectly can have serious ramifications, so
we need to employ over-the-top input validation and data verification
mechanisms.
```

Notice that we didn't provide an operative revision this time. That's because when no operative revision is specified, Subversion assumes a default operative revision that's the same as the peg revision.

As you can see, the output from our operation appears to be correct. The text even mentions frabbing naggily worts, so this is almost certainly the file which describes the software now called Frabnaggilywort. In fact, we can verify this using the combination of an explicit peg revision and explicit operative revision. We know that in `HEAD`, the Frabnaggilywort project is located in the `frabnaggilywort` directory. So we specify that we want to see how the line of history identified in `HEAD` as the path `frabnaggilywort/IDEA` looked in revision 1.

```
$ svn cat -r 1 frabnaggilywort/IDEA@HEAD
The idea behind this project is to come up with a piece of software
that can frab a naggily wort.  Frabbing naggily worts is tricky
business, and doing it incorrectly can have serious ramifications, so
we need to employ over-the-top input validation and data verification
mechanisms.
```

And the peg and operative revisions need not be so trivial, either. For example, say `frabnaggilywort` had been deleted from `HEAD`, but we know it existed in revision 20, and we want to see the diffs for its `IDEA` file between revisions 4 and 10. We can use the peg revision 20 in conjunction with the URL that would have held Frabnaggilywort's `IDEA` file in revision 20, and then use 4 and 10 as our operative revision range.

```
$ svn diff -r 4:10 http://svn.red-bean.com/projects/frabnaggilywort/IDEA@20
Index: frabnaggilywort/IDEA
===================================================================
--- frabnaggilywort/IDEA (revision 4)
+++ frabnaggilywort/IDEA (revision 10)
@@ -1,5 +1,5 @@
-The idea behind this project is to come up with a piece of software
-that can frab a naggily wort.  Frabbing naggily worts is tricky
-business, and doing it incorrectly can have serious ramifications, so
```

```
-we need to employ over-the-top input validation and data verification
-mechanisms.
+The idea behind this project is to come up with a piece of
+client-server software that can remotely frab a naggily wort.
+Frabbing naggily worts is tricky business, and doing it incorrectly
+can have serious ramifications, so we need to employ over-the-top
+input validation and data verification mechanisms.
```

Fortunately, most folks aren't faced with such complex situations. But when you are, remember that peg revisions are that extra hint Subversion needs to clear up ambiguity.

Network Model

At some point, you're going to need to understand how your Subversion client communicates with its server. Subversion's networking layer is abstracted, meaning that Subversion clients exhibit the same general behaviors no matter what sort of server they are operating against. Whether speaking the HTTP protocol (`http://`) with the Apache HTTP Server or speaking the custom Subversion protocol (`svn://`) with **svnserve**, the basic network model is the same. In this section, we'll explain the basics of that network model, including how Subversion manages authentication and authorization matters.

Requests and Responses

The Subversion client spends most of its time managing working copies. When it needs information from a remote repository, however, it makes a network request, and the server responds with an appropriate answer. The details of the network protocol are hidden from the user—the client attempts to access a URL, and depending on the URL scheme, a particular protocol is used to contact the server (see Repository URLs).



Run **svn --version** to see which URL schemes and protocols the client knows how to use.

When the server process receives a client request, it often demands that the client identify itself. It issues an authentication challenge to the client, and the client responds by providing *credentials* back to the server. Once authentication is complete, the server responds with the original information the client asked for. Notice that this system is different from systems like CVS, where the client pre-emptively offers credentials (“logs in”) to the server before ever making a request. In Subversion, the server “pulls” credentials by challenging the client at the appropriate moment, rather than the client “pushing” them. This makes certain operations more elegant. For example, if a server is configured to allow anyone in the world to read a repository, then the server will never issue an authentication challenge when a client attempts to **svn checkout**.

If the particular network requests issued by the client result in a new revision being created in the repository, (e.g. **svn commit**), then Subversion uses the authenticated username associated with those requests as the author of the revision. That is, the authenticated user's name is stored as the value of the `svn:author` property on the new revision (see the section called “Subversion properties”). If the client was not authenticated (in other words, the server never issued an authentication challenge), then the revision's `svn:author` property is empty.

Client Credentials Caching

Many servers are configured to require authentication on every request. This would be a big

annoyance to users, if they were forced to type their passwords over and over again. Fortunately, the Subversion client has a remedy for this—a built-in system for caching authentication credentials on disk. By default, whenever the command-line client successfully responds to a server's authentication challenge, it saves the credentials in the user's private runtime configuration area (`~/.subversion/auth/` on Unix-like systems or `%APPDATA%/Subversion/auth/` on Windows; see the section called “Runtime Configuration Area” for more details about the runtime configuration system). Successful credentials are cached on disk, keyed on a combination of the server's hostname, port, and authentication realm.

When the client receives an authentication challenge, it first looks for the appropriate credentials in the user's disk cache. If seemingly suitable credentials are not present, or if the cached credentials ultimately fail to authenticate, then the client will, by default, fall back to prompting the user for the necessary information.

The security-conscious reader will suspect immediately that there is reason for concern here. “Caching passwords on disk? That's terrible! You should never do that!”

The Subversion developers recognize the legitimacy of such concerns, and so Subversion works with available mechanisms provided by the operating system and environment to try to minimize the risk of leaking this information. Here's a breakdown of what this means for users on the most common platforms:

- On Windows 2000 and later, the Subversion client uses standard Windows cryptography services to encrypt the password on disk. Because the encryption key is managed by Windows and is tied to the user's own login credentials, only the user can decrypt the cached password. (Note that if the user's Windows account password is reset by an administrator, all of the cached passwords become undecipherable. The Subversion client will behave as if they don't exist, prompting for passwords when required.)

- Similarly, on Mac OS X, the Subversion client stores all repository passwords in the login keyring (managed by the Keychain service), which is protected by the user's account password. User preference settings can impose additional policies, such as requiring the user's account password be entered each time the Subversion password is used.

- For other Unix-like operating systems, no standard “keychain” services exist. However, the `auth/` caching area is still permission-protected so that only the user (owner) can read data from it, not the world at large. The operating system's own file permissions protect the passwords.

Of course, for the truly paranoid, none of these mechanisms meets the test of perfection. So for those folks willing to sacrifice convenience for the ultimate security, Subversion provides various ways of disabling its credentials caching system altogether.

To disable caching for a single command, pass the `--no-auth-cache` option:

```
$ svn commit -F log_msg.txt --no-auth-cache
Authentication realm: <svn://host.example.com:3690> example realm
Username:  joe
Password for 'joe':

Adding         newfile
Transmitting file data .
Committed revision 2324.

# password was not cached, so a second commit still prompts us
```

```
$ svn delete newfile
$ svn commit -F new_msg.txt
Authentication realm: <svn://host.example.com:3690> example realm
Username:  joe
…
```

Or, if you want to disable credential caching permanently, you can edit the `config` file in your runtime configuration area, and set the `store-auth-creds` option to `no`. This will prevent the storing of credentials used in any Subversion interactions you perform on the affected computer. This can be extended to cover all users on the computer, too, by modifying the system-wide runtime configuration area (described in the section called “Configuration Area Layout”).

```
[auth]
store-auth-creds = no
```

Sometimes users will want to remove specific credentials from the disk cache. To do this, you need to navigate into the `auth/` area and manually delete the appropriate cache file. Credentials are cached in individual files; if you look inside each file, you will see keys and values. The `svn:realmstring` key describes the particular server realm that the file is associated with:

```
$ ls ~/.subversion/auth/svn.simple/
5671adf2865e267db74f09ba6f872c28
3893ed123b39500bca8a0b382839198e
5c3c22968347b390f349ff340196ed39

$ cat ~/.subversion/auth/svn.simple/5671adf2865e267db74f09ba6f872c28

K 8
username
V 3
joe
K 8
password
V 4
blah
K 15
svn:realmstring
V 45
<https://svn.domain.com:443> Joe's repository
END
```

Once you have located the proper cache file, just delete it.

One last word about **svn**'s authentication behavior, specifically regarding the `--username` and `--password` options. Many client subcommands accept these options, but it is important to understand using these options does *not* automatically send credentials to the server. As discussed earlier, the server “pulls” credentials from the client when it deems necessary; the client cannot “push” them at will. If a username and/or password are passed as options, they will only be presented to the server if the server requests them. [12] These options are typically used to authenticate as a different user than Subversion would have chosen by default (such as your system login name), or when trying to avoid interactive prompting (such as when call-

[12] Again, a common mistake is to misconfigure a server so that it never issues an authentication challenge. When users pass `--username` and `--password` options to the client, they're surprised to see that they're never used, i.e. new revisions still appear to have been committed anonymously!

ing **svn** from a script).

Here is a final summary that describes how a Subversion client behaves when it receives an authentication challenge.

1. First, the client checks whether the user specified any credentials as command-line options (`--username` and/or `--password`). If not, or if these options fail to authenticate successfully, then

2. the client looks up the server's hostname, port, and realm in the runtime `auth/` area, to see if the user already has the appropriate credentials cached. If not, or if the cached credentials fail to authenticate, then

3. finally, the client resorts to prompting the user (unless instructed not to do so via the `--non-interactive` option or its client-specific equivalents).

If the client successfully authenticates by any of the methods listed above, it will attempt to cache the credentials on disk (unless the user has disabled this behavior, as mentioned earlier).

Chapter 4. Branching and Merging

"#### (It is upon the Trunk that a gentleman works.)"

—Confucius

Branching, tagging, and merging are concepts common to almost all version control systems. If you're not familiar with these ideas, we provide a good introduction in this chapter. If you are familiar, then hopefully you'll find it interesting to see how Subversion implements these ideas.

Branching is a fundamental part of version control. If you're going to allow Subversion to manage your data, then this is a feature you'll eventually come to depend on. This chapter assumes that you're already familiar with Subversion's basic concepts (Chapter 1, *Fundamental Concepts*).

What's a Branch?

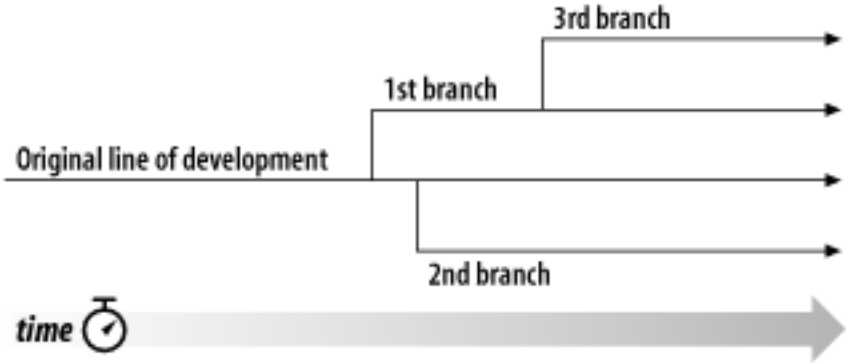
Suppose it's your job to maintain a document for a division in your company, a handbook of some sort. One day a different division asks you for the same handbook, but with a few parts "tweaked" for them, since they do things slightly differently.

What do you do in this situation? You do the obvious thing: you make a second copy of your document, and begin maintaining the two copies separately. As each department asks you to make small changes, you incorporate them into one copy or the other.

You often want to make the same change to both copies. For example, if you discover a typo in the first copy, it's very likely that the same typo exists in the second copy. The two documents are almost the same, after all; they only differ in small, specific ways.

This is the basic concept of a *branch*—namely, a line of development that exists independently of another line, yet still shares a common history if you look far enough back in time. A branch always begins life as a copy of something, and moves on from there, generating its own history (see Figure 4.1, "Branches of development").

Figure 4.1. Branches of development



Subversion has commands to help you maintain parallel branches of your files and directories. It allows you to create branches by copying your data, and remembers that the copies are related to one another. It also helps you duplicate changes from one branch to another. Finally, it can make portions of your working copy reflect different branches, so that you can "mix and

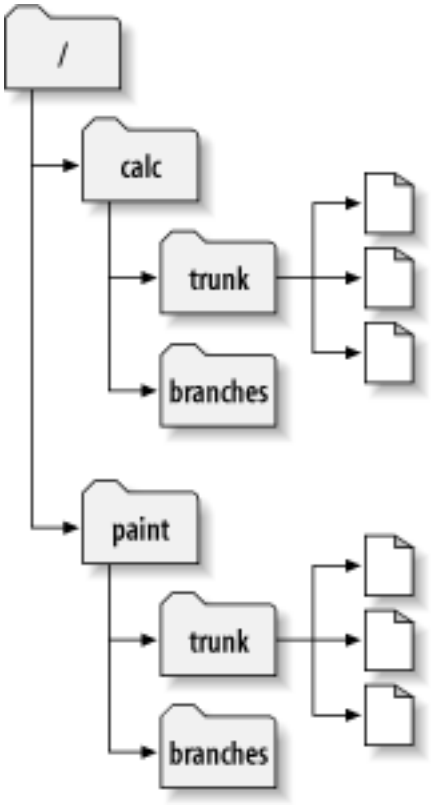
match” different lines of development in your daily work.

Using Branches

At this point, you should understand how each commit creates an entire new filesystem tree (called a “revision”) in the repository. If not, go back and read about revisions in the section called “Revisions”.

For this chapter, we'll go back to the same example from Chapter 1, *Fundamental Concepts*. Remember that you and your collaborator, Sally, are sharing a repository that contains two projects, `paint` and `calc`. Notice that in Figure 4.2, “Starting repository layout”, however, each project directory now contains subdirectories named `trunk` and `branches`. The reason for this will soon become clear.

Figure 4.2. Starting repository layout



As before, assume that Sally and you both have working copies of the “calc” project. Specifically, you each have a working copy of `/calc/trunk`. All the files for the project are in this subdirectory rather than in `/calc` itself, because your team has decided that `/calc/trunk` is where the “main line” of development is going to take place.

Let's say that you've been given the task of implementing radical new project feature. It will take a long time to write, and will affect all the files in the project. The problem here is that you don't want to interfere with Sally, who is in the process of fixing small bugs here and there. She's depending on the fact that the latest version of the project (in `/calc/trunk`) is always usable. If you start committing your changes bit-by-bit, you'll surely break things for Sally.

One strategy is to crawl into a hole: you and Sally can stop sharing information for a week or two. That is, start gutting and reorganizing all the files in your working copy, but don't commit or update until you're completely finished with the task. There are a number of problems with this, though. First, it's not very safe. Most people like to save their work to the repository frequently, should something bad accidentally happen to their working copy. Second, it's not very flexible. If you do your work on different computers (perhaps you have a working copy of `/calc/trunk` on two different machines), you'll need to manually copy your changes back and forth, or just do all the work on a single computer. By that same token, it's difficult to share your changes-in-progress with anyone else. A common software development “best practice” is to allow your peers to review your work as you go. If nobody sees your intermediate commits, you lose potential feedback. Finally, when you're finished with all your changes, you might find it very difficult to re-merge your final work with the rest of the company's main body of code. Sally (or others) may have made many other changes in the repository that are difficult to incorporate into your working copy—especially if you run **svn update** after weeks of isolation.

The better solution is to create your own branch, or line of development, in the repository. This allows you to save your half-broken work frequently without interfering with others, yet you can still selectively share information with your collaborators. You'll see exactly how this works later on.

Creating a Branch

Creating a branch is very simple—you make a copy of the project in the repository using the **svn copy** command. Subversion is not only able to copy single files, but whole directories as well. In this case, you want to make a copy of the `/calc/trunk` directory. Where should the new copy live? Wherever you wish—it's a matter of project policy. Let's say that your team has a policy of creating branches in the `/calc/branches` area of the repository, and you want to name your branch `my-calc-branch`. You'll want to create a new directory, `/calc/branches/my-calc-branch`, which begins its life as a copy of `/calc/trunk`.

There are two different ways to make a copy. We'll demonstrate the messy way first, just to make the concept clear. To begin, check out a working copy of the project's root directory, `/calc`:

```
$ svn checkout http://svn.example.com/repos/calc bigwc
A  bigwc/trunk/
A  bigwc/trunk/Makefile
A  bigwc/trunk/integer.c
A  bigwc/trunk/button.c
A  bigwc/branches/
Checked out revision 340.
```

Making a copy is now simply a matter of passing two working-copy paths to the **svn copy** command:

```
$ cd bigwc
$ svn copy trunk branches/my-calc-branch
$ svn status
A  +   branches/my-calc-branch
```

In this case, the **svn copy** command recursively copies the `trunk` working directory to a new working directory, `branches/my-calc-branch`. As you can see from the **svn status** command, the new directory is now scheduled for addition to the repository. But also notice the “+” sign next to the letter A. This indicates that the scheduled addition is a *copy* of something, not

something new. When you commit your changes, Subversion will create `/calc/branches/my-calc-branch` in the repository by copying `/calc/trunk`, rather than resending all of the working copy data over the network:

```
$ svn commit -m "Creating a private branch of /calc/trunk."
Adding         branches/my-calc-branch
Committed revision 341.
```

And now here's the easier method of creating a branch, which we should have told you about in the first place: **svn copy** is able to operate directly on two URLs.

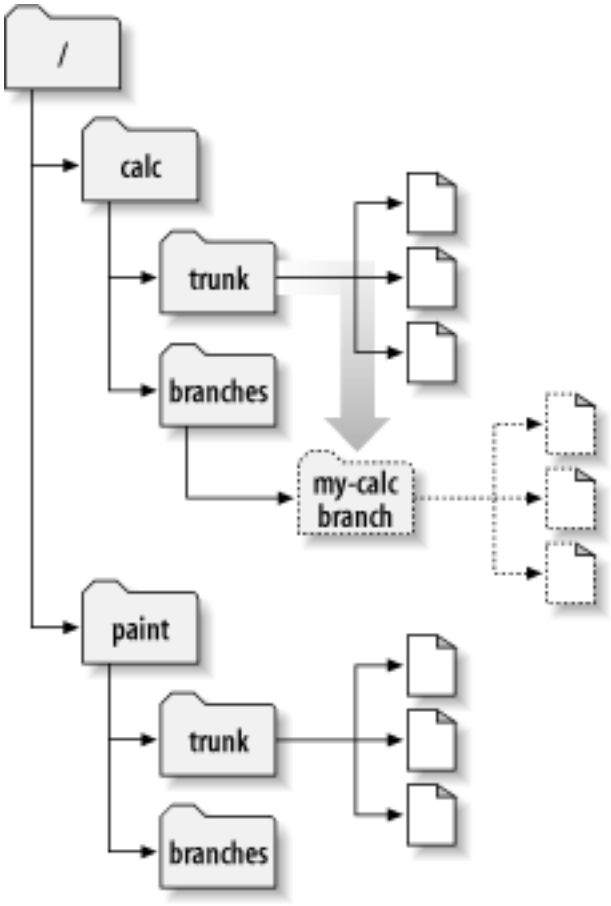
```
$ svn copy http://svn.example.com/repos/calc/trunk \
           http://svn.example.com/repos/calc/branches/my-calc-branch \
      -m "Creating a private branch of /calc/trunk."

Committed revision 341.
```

From the repository's point of view, there's really no difference between these two methods. Both procedures create a new directory in revision 341, and the new directory is a copy of `/calc/trunk`. This is shown in Figure 4.3, “Repository with new copy”. Notice that the second method, however, performs an *immediate* commit in constant time. [1] It's an easier procedure, because it doesn't require you to check out a large portion of the repository. In fact, this technique doesn't even require you to have a working copy at all. This is the way most users create branches.

Figure 4.3. Repository with new copy

[1] Subversion does not support copying between different repositories. When using URLs with **svn copy** or **svn move**, you can only copy items within the same repository.



Cheap Copies

Subversion's repository has a special design. When you copy a directory, you don't need to worry about the repository growing huge—Subversion doesn't actually duplicate any data. Instead, it creates a new directory entry that points to an *existing* tree. If you're a Unix user, this is the same concept as a hard-link. As further changes are made to files and directories beneath the copied directory, Subversion continues to employ this hard-link concept where it can. It only duplicates data when it is necessary to disambiguate different versions of objects.

This is why you'll often hear Subversion users talk about “cheap copies”. It doesn't matter how large the directory is—it takes a very tiny, constant amount of time to make a copy of it. In fact, this feature is the basis of how commits work in Subversion: each revision is a “cheap copy” of the previous revision, with a few items lazily changed within. (To read more about this, visit Subversion's website and read about the “bubble up” method in Subversion's design documents.)

Of course, these internal mechanics of copying and sharing data are hidden from the user, who simply sees copies of trees. The main point here is that copies are cheap, both in time and space. If you create a branch entirely within the repository (by running **svn copy URL1 URL2**), it's a quick, constant-time operation. Make branches as often as you want.

Working with Your Branch

Now that you've created a branch of the project, you can check out a new working copy to start using it:

```
$ svn checkout http://svn.example.com/repos/calc/branches/my-calc-branch
A  my-calc-branch/Makefile
A  my-calc-branch/integer.c
A  my-calc-branch/button.c
Checked out revision 341.
```

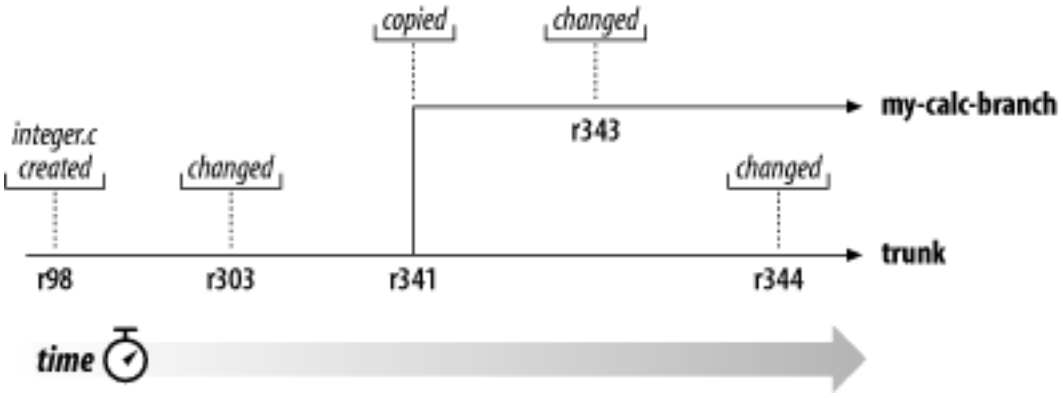
There's nothing special about this working copy; it simply mirrors a different directory in the repository. When you commit changes, however, Sally won'tsee them when she updates, because her working copy is of `/calc/trunk`. (Be sure to read the section called “Traversing Branches” later in this chapter: the **svn switch** command is an alternate way of creating a working copy of a branch.)

Let's pretend that a week goes by, and the following commits happen:

- You make a change to `/calc/branches/my-calc-branch/button.c`, which creates revision 342.
- You make a change to `/calc/branches/my-calc-branch/integer.c`, which creates revision 343.
- Sally makes a change to `/calc/trunk/integer.c`, which creates revision 344.

There are now two independent lines of development, shown in Figure 4.4, “The branching of one file's history”, happening on `integer.c`.

Figure 4.4. The branching of one file's history



Things get interesting when you look at the history of changes made to your copy of `integer.c`:

```
$ pwd
/home/user/my-calc-branch
```

```
$ svn log -v integer.c
------------------------------------------------------------------------
r343 | user | 2002-11-07 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
Changed paths:
   M /calc/branches/my-calc-branch/integer.c

* integer.c:  frozzled the wazjub.

------------------------------------------------------------------------
r341 | user | 2002-11-03 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
Changed paths:
   A /calc/branches/my-calc-branch (from /calc/trunk:340)

Creating a private branch of /calc/trunk.

------------------------------------------------------------------------
r303 | sally | 2002-10-29 21:14:35 -0600 (Tue, 29 Oct 2002) | 2 lines
Changed paths:
   M /calc/trunk/integer.c

* integer.c:  changed a docstring.

------------------------------------------------------------------------
r98 | sally | 2002-02-22 15:35:29 -0600 (Fri, 22 Feb 2002) | 2 lines
Changed paths:
   M /calc/trunk/integer.c

* integer.c:  adding this file to the project.

------------------------------------------------------------------------
```

Notice that Subversion is tracing the history of your branch's `integer.c` all the way back through time, even traversing the point where it was copied. It shows the creation of the branch as an event in the history, because `integer.c` was implicitly copied when all of `/calc/trunk/` was copied. Now look what happens when Sally runs the same command on her copy of the file:

```
$ pwd
/home/sally/calc

$ svn log -v integer.c
------------------------------------------------------------------------
r344 | sally | 2002-11-07 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
Changed paths:
   M /calc/trunk/integer.c

* integer.c:  fix a bunch of spelling errors.

------------------------------------------------------------------------
r303 | sally | 2002-10-29 21:14:35 -0600 (Tue, 29 Oct 2002) | 2 lines
Changed paths:
   M /calc/trunk/integer.c

* integer.c:  changed a docstring.

------------------------------------------------------------------------
r98 | sally | 2002-02-22 15:35:29 -0600 (Fri, 22 Feb 2002) | 2 lines
Changed paths:
   M /calc/trunk/integer.c

* integer.c:  adding this file to the project.
```

Sally sees her own revision 344 change, but not the change you made in revision 343. As far as Subversion is concerned, these two commits affected different files in different repository locations. However, Subversion *does* show that the two files share a common history. Before the branch-copy was made in revision 341, they used to be the same file. That's why you and Sally both see the changes made in revisions 303 and 98.

The Key Concepts Behind Branches

There are two important lessons that you should remember from this section. First, Subversion has no internal concept of a branch—it only knows how to make copies. When you copy a directory, the resulting directory is only a “branch” because *you* attach that meaning to it. You may think of the directory differently, or treat it differently, but to Subversion it's just an ordinary directory that happens to carry some extra historical information. Second, because of this copy mechanism, Subversion's branches exist as *normal filesystem directories* in the repository. This is different from other version control systems, where branches are typically defined by adding extra-dimensional “labels” to collections of files.

Copying Changes Between Branches

Now you and Sally are working on parallel branches of the project: you're working on a private branch, and Sally is working on the *trunk*, or main line of development.

For projects that have a large number of contributors, it's common for most people to have working copies of the trunk. Whenever someone needs to make a long-running change that is likely to disrupt the trunk, a standard procedure is to create a private branch and commit changes there until all the work is complete.

So, the good news is that you and Sally aren't interfering with each other. The bad news is that it's very easy to drift *too* far apart. Remember that one of the problems with the “crawl in a hole” strategy is that by the time you're finished with your branch, it may be near-impossible to merge your changes back into the trunk without a huge number of conflicts.

Instead, you and Sally might continue to share changes as you work. It's up to you to decide which changes are worth sharing; Subversion gives you the ability to selectively “copy” changes between branches. And when you're completely finished with your branch, your entire set of branch changes can be copied back into the trunk.

Copying Specific Changes

In the previous section, we mentioned that both you and Sally made changes to `integer.c` on different branches. If you look at Sally's log message for revision 344, you can see that she fixed some spelling errors. No doubt, your copy of the same file still has the same spelling errors. It's likely that your future changes to this file will be affecting the same areas that have the spelling errors, so you're in for some potential conflicts when you merge your branch someday. It's better, then, to receive Sally's change now, *before* you start working too heavily in the same places.

It's time to use the **svn merge** command. This command, it turns out, is a very close cousin to the **svn diff** command (which you read about in Chapter 2, *Basic Usage*). Both commands are able to compare any two objects in the repository and describe the differences. For example, you can ask **svn diff** to show you the exact change made by Sally in revision 344:

```
$ svn diff -c 344 http://svn.example.com/repos/calc/trunk

Index: integer.c
===================================================================
--- integer.c (revision 343)
+++ integer.c (revision 344)
@@ -147,7 +147,7 @@
     case 6:  sprintf(info->operating_system, "HPFS (OS/2 or NT)"); break;
     case 7:  sprintf(info->operating_system, "Macintosh"); break;
     case 8:  sprintf(info->operating_system, "Z-System"); break;
-    case 9:  sprintf(info->operating_system, "CPM"); break;
+    case 9:  sprintf(info->operating_system, "CP/M"); break;
     case 10:  sprintf(info->operating_system, "TOPS-20"); break;
     case 11:  sprintf(info->operating_system, "NTFS (Windows NT)"); break;
     case 12:  sprintf(info->operating_system, "QDOS"); break;
@@ -164,7 +164,7 @@
     low = (unsigned short) read_byte(gzfile);  /* read LSB */
     high = (unsigned short) read_byte(gzfile); /* read MSB */
     high = high << 8;  /* interpret MSB correctly */
-    total = low + high; /* add them togethe for correct total */
+    total = low + high; /* add them together for correct total */

     info->extra_header = (unsigned char *) my_malloc(total);
     fread(info->extra_header, total, 1, gzfile);
@@ -241,7 +241,7 @@
      Store the offset with ftell() ! */

   if ((info->data_offset = ftell(gzfile))== -1) {
-    printf("error: ftell() retturned -1.\n");
+    printf("error: ftell() returned -1.\n");
     exit(1);
   }

@@ -249,7 +249,7 @@
   printf("I believe start of compressed data is %u\n", info->data_offset);
   #endif

-  /* Set postion eight bytes from the end of the file. */
+  /* Set position eight bytes from the end of the file. */

   if (fseek(gzfile, -8, SEEK_END)) {
     printf("error: fseek() returned non-zero\n");
```

The **svn merge** command is almost exactly the same. Instead of printing the differences to your terminal, however, it applies them directly to your working copy as *local modifications*:

```
$ svn merge -c 344 http://svn.example.com/repos/calc/trunk
U  integer.c

$ svn status
M  integer.c
```

The output of **svn merge** shows that your copy of `integer.c` was patched. It now contains Sally's change—the change has been “copied” from the trunk to your working copy of your private branch, and now exists as a local modification. At this point, it's up to you to review the local modification and make sure it works correctly.

In another scenario, it's possible that things may not have gone so well, and that `integer.c` may have entered a conflicted state. You might need to resolve the conflict using standard pro-

cedures (see Chapter 2, *Basic Usage*), or if you decide that the merge was a bad idea altogether, simply give up and **svn revert** the local change.

But assuming that you've reviewed the merged change, you can **svn commit** the change as usual. At that point, the change has been merged into your repository branch. In version control terminology, this act of copying changes between branches is commonly called *porting* changes.

When you commit the local modification, make sure your log message mentions that you're porting a specific change from one branch to another. For example:

```
$ svn commit -m "integer.c: ported r344 (spelling fixes) from trunk."
Sending        integer.c
Transmitting file data .
Committed revision 360.
```

As you'll see in the next sections, this is a very important “best practice” to follow.

Why Not Use Patches Instead?

A question may be on your mind, especially if you're a Unix user: why bother to use **svn merge** at all? Why not simply use the operating system's **patch** command to accomplish the same job? For example:

```
$ svn diff -c 344 http://svn.example.com/repos/calc/trunk > patchfile
$ patch -p0  < patchfile
Patching file integer.c using Plan A...
Hunk #1 succeeded at 147.
Hunk #2 succeeded at 164.
Hunk #3 succeeded at 241.
Hunk #4 succeeded at 249.
done
```

In this particular case, yes, there really is no difference. But **svn merge** has special abilities that surpass the **patch** program. The file format used by **patch** is quite limited; it's only able to tweak file contents. There's no way to represent changes to *trees*, such as the addition, removal, or renaming of files and directories. Nor can the **patch** program notice changes to properties. If Sally's change had, say, added a new directory, the output of **svn diff** wouldn't have mentioned it at all. **svn diff** only outputs the limited patch-format, so there are some ideas it simply can't express. The **svn merge** command, however, can express changes in tree structure and properties by directly applying them to your working copy.

A word of warning: while **svn diff** and **svn merge** are very similar in concept, they do have different syntax in many cases. Be sure to read about them in Chapter 9, *Subversion Complete Reference* for details, or ask **svn help**. For example, **svn merge** requires a working-copy path as a target, i.e. a place where it should apply the tree-changes. If the target isn't specified, it assumes you are trying to perform one of the following common operations:

1. You want to merge directory changes into your current working directory.
2. You want to merge the changes in a specific file into a file by the same name which exists in your current working directory.

If you are merging a directory and haven't specified a target path, **svn merge** assumes the first case above and tries to apply the changes into your current directory. If you are merging a file, and that file (or a file by the same name) exists in your current working directory, **svn merge** assumes the second case and tries to apply the changes to a local file with the same name.

If you want changes applied somewhere else, you'll need to say so. For example, if you're sitting in the parent directory of your working copy, you'll have to specify the target directory to receive the changes:

```
$ svn merge -c 344 http://svn.example.com/repos/calc/trunk my-calc-branch
U   my-calc-branch/integer.c
```

The Key Concept Behind Merging

You've now seen an example of the **svn merge** command, and you're about to see several more. If you're feeling confused about exactly how merging works, you're not alone. Many users (especially those new to version control) are initially perplexed about the proper syntax of the command, and about how and when the feature should be used. But fear not, this command is actually much simpler than you think! There's a very easy technique for understanding exactly how **svn merge** behaves.

The main source of confusion is the *name* of the command. The term “merge” somehow denotes that branches are combined together, or that there's some sort of mysterious blending of data going on. That's not the case. A better name for the command might have been **svn diff-and-apply**, because that's all that happens: two repository trees are compared, and the differences are applied to a working copy.

The command takes three arguments:

1. An initial repository tree (often called the *left side* of the comparison),

2. A final repository tree (often called the *right side* of the comparison),

3. A working copy to accept the differences as local changes (often called the *target* of the merge).

Once these three arguments are specified, the two trees are compared, and the resulting differences are applied to the target working copy as local modifications. When the command is done, the results are no different than if you had hand-edited the files, or run various **svn add** or **svn delete** commands yourself. If you like the results, you can commit them. If you don't like the results, you can simply **svn revert** all of the changes.

The syntax of **svn merge** allows you to specify the three necessary arguments rather flexibly. Here are some examples:

```
$ svn merge http://svn.example.com/repos/branch1@150 \
            http://svn.example.com/repos/branch2@212 \
            my-working-copy

$ svn merge -r 100:200 http://svn.example.com/repos/trunk my-working-copy

$ svn merge -r 100:200 http://svn.example.com/repos/trunk
```

The first syntax lays out all three arguments explicitly, naming each tree in the form *URL@REV* and naming the working copy target. The second syntax can be used as a shorthand for situations when you're comparing two different revisions of the same URL. The last syntax shows how the working-copy argument is optional; if omitted, it defaults to the current directory.

Best Practices for Merging

Tracking Merges Manually

Merging changes sounds simple enough, but in practice it can become a headache. The problem is that if you repeatedly merge changes from one branch to another, you might accidentally merge the same change *twice*. When this happens, sometimes things will work fine. When patching a file, Subversion typically notices if the file already has the change, and does nothing. But if the already-existing change has been modified in any way, you'll get a conflict.

Ideally, your version control system should prevent the double-application of changes to a branch. It should automatically remember which changes a branch has already received, and be able to list them for you. It should use this information to help automate merges as much as possible.

Unfortunately, Subversion is not such a system; it does not yet record any information about merge operations. [2] When you commit local modifications, the repository has no idea whether those changes came from running **svn merge**, or from just hand-editing the files.

What does this mean to you, the user? It means that until the day Subversion grows this feature, you'll have to track merge information yourself. The best place to do this is in the commit log-message. As demonstrated in prior examples, it's recommended that your log-message mention a specific revision number (or range of revisions) that are being merged into your branch. Later on, you can run **svn log** to review which changes your branch already contains. This will allow you to carefully construct a subsequent **svn merge** command that won't be redundant with previously ported changes.

In the next section, we'll show some examples of this technique in action.

Previewing Merges

First, always remember to do your merge into a working copy that has *no* local edits and has been recently updated. If your working copy isn't “clean” in these ways, you can run into some headaches.

Assuming your working copy is tidy, merging isn't a particularly high-risk operation. If you get the merge wrong the first time, simply **svn revert** the changes and try again.

If you've merged into a working copy that already has local modifications, the changes applied by a merge will be mixed with your pre-existing ones, and running **svn revert** is no longer an option. The two sets of changes may be impossible to separate.

In cases like this, people take comfort in being able to predict or examine merges before they happen. One simple way to do that is to run **svn diff** with the same arguments you plan to pass to **svn merge**, as we already showed in our first example of merging. Another method of previewing is to pass the `--dry-run` option to the merge command:

[2] However, at the time of writing, this feature is being worked on!

```
$ svn merge --dry-run -c 344 http://svn.example.com/repos/calc/trunk
U  integer.c

$ svn status
#  nothing printed, working copy is still unchanged.
```

The `--dry-run` option doesn't actually apply any local changes to the working copy. It only shows status codes that *would* be printed in a real merge. It's useful for getting a “high level” preview of the potential merge, for those times when running **svn diff** gives too much detail.

Merge Conflicts

Just like the **svn update** command, **svn merge** applies changes to your working copy. And therefore it's also capable of creating conflicts. The conflicts produced by **svn merge**, however, are sometimes different, and this section explains those differences.

To begin with, assume that your working copy has no local edits. When you **svn update** to a particular revision, the changes sent by the server will always apply “cleanly” to your working copy. The server produces the delta by comparing two trees: a virtual snapshot of your working copy, and the revision tree you're interested in. Because the left-hand side of the comparison is exactly equal to what you already have, the delta is guaranteed to correctly convert your working copy into the right-hand tree.

But **svn merge** has no such guarantees and can be much more chaotic: the user can ask the server to compare *any* two trees at all, even ones that are unrelated to the working copy! This means there's large potential for human error. Users will sometimes compare the wrong two trees, creating a delta that doesn't apply cleanly. **svn merge** will do its best to apply as much of the delta as possible, but some parts may be impossible. Just as the Unix **patch** command sometimes complains about “failed hunks”, **svn merge** will complain about “skipped targets”:

```
$ svn merge -r 1288:1351 http://svn.example.com/repos/branch
U  foo.c
U  bar.c
Skipped missing target: 'baz.c'
U  glub.c
C  glorb.h

$
```

In the previous example it might be the case that `baz.c` exists in both snapshots of the branch being compared, and the resulting delta wants to change the file's contents, but the file doesn't exist in the working copy. Whatever the case, the “skipped” message means that the user is most likely comparing the wrong two trees; they're the classic sign of user error. When this happens, it's easy to recursively revert all the changes created by the merge (**svn revert --recursive**), delete any unversioned files or directories left behind after the revert, and re-run **svn merge** with different arguments.

Also notice that the previous example shows a conflict happening on `glorb.h`. We already stated that the working copy has no local edits: how can a conflict possibly happen? Again, because the user can use **svn merge** to define and apply any old delta to the working copy, that delta may contain textual changes that don't cleanly apply to a working file, even if the file has no local modifications.

Another small difference between **svn update** and **svn merge** are the names of the full-text files created when a conflict happens. In the section called “Resolve Conflicts (Merging Others' Changes)”, we saw that an update produces files named `filename.mine`, `file-`

name.rOLDREV, and filename.rNEWREV. When **svn merge** produces a conflict, though, it creates three files named filename.working, filename.left, and filename.right. In this case, the terms “left” and “right” are describing which side of the double-tree comparison the file came from. In any case, these differing names will help you distinguish between conflicts that happened as a result of an update versus ones that happened as a result of a merge.

Noticing or Ignoring Ancestry

When conversing with a Subversion developer, you might very likely hear reference to the term *ancestry*. This word is used to describe the relationship between two objects in a repository: if they're related to each other, then one object is said to be an ancestor of the other.

For example, suppose you commit revision 100, which includes a change to a file foo.c. Then foo.c@99 is an “ancestor” of foo.c@100. On the other hand, suppose you commit the deletion of foo.c in revision 101, and then add a new file by the same name in revision 102. In this case, foo.c@99 and foo.c@102 may appear to be related (they have the same path), but in fact are completely different objects in the repository. They share no history or “ancestry”.

The reason for bringing this up is to point out an important difference between **svn diff** and **svn merge**. The former command ignores ancestry, while the latter command is quite sensitive to it. For example, if you asked **svn diff** to compare revisions 99 and 102 of foo.c, you would see line-based diffs; the diff command is blindly comparing two paths. But if you asked **svn merge** to compare the same two objects, it would notice that they're unrelated and first attempt to delete the old file, then add the new file; the output would indicate a deletion followed by an add:

```
D  foo.c
A  foo.c
```

Most merges involve comparing trees that are ancestrally related to one another, and therefore **svn merge** defaults to this behavior. Occasionally, however, you may want the merge command to compare two unrelated trees. For example, you may have imported two source-code trees representing different vendor releases of a software project (see the section called “Vendor branches”). If you asked **svn merge** to compare the two trees, you'd see the entire first tree being deleted, followed by an add of the entire second tree! In these situations, you'll want **svn merge** to do a path-based comparison only, ignoring any relations between files and directories. Add the --ignore-ancestry option to your merge command, and it will behave just like **svn diff**. (And conversely, the --notice-ancestry option will cause **svn diff** to behave like the merge command.)

Merges and Moves

A common desire is to refactor source code, especially in Java-based software projects. Files and directories are shuffled around and renamed, often causing great disruption to everyone working on the project. Sounds like a perfect case to use a branch, doesn't it? Just create a branch, shuffle things around, then merge the branch back to the trunk, right?

Alas, this scenario doesn't work so well right now, and is considered one of Subversion's current weak spots. The problem is that Subversion's **update** command isn't as robust as it should be, particularly when dealing with copy and move operations.

When you use **svn copy** to duplicate a file, the repository remembers where the new file came from, but it fails to transmit that information to the client which is running **svn update** or **svn merge**. Instead of telling the client, “Copy that file you already have to this new location”, it instead sends down an entirely new file. This can lead to problems, especially because the

same thing happens with renamed files. A lesser-known fact about Subversion is that it lacks “true renames”—the **svn move** command is nothing more than an aggregation of **svn copy** and **svn delete**.

For example, suppose that while working on your private branch, you rename `integer.c` to `whole.c`. Effectively you've created a new file in your branch that is a copy of the original file, and deleted the original file. Meanwhile, back on `trunk`, Sally has committed some improvements to `integer.c`. Now you decide to merge your branch to the trunk:

```
$ cd calc/trunk

$ svn merge -r 341:405 http://svn.example.com/repos/calc/branches/my-calc-branch
D   integer.c
A   whole.c
```

This doesn't look so bad at first glance, but it's also probably not what you or Sally expected. The merge operation has deleted the latest version of `integer.c` file (the one containing Sally's latest changes), and blindly added your new `whole.c` file—which is a duplicate of the *older* version of `integer.c`. The net effect is that merging your “rename” to the branch has removed Sally's recent changes from the latest revision!

This isn't true data-loss; Sally's changes are still in the repository's history, but it may not be immediately obvious that this has happened. The moral of this story is that until Subversion improves, be very careful about merging copies and renames from one branch to another.

Common Use-Cases

There are many different uses for branching and **svn merge**, and this section describes the most common ones you're likely to run into.

Merging a Whole Branch to Another

To complete our running example, we'll move forward in time. Suppose several days have passed, and many changes have happened on both the trunk and your private branch. Suppose that you've finished working on your private branch; the feature or bug fix is finally complete, and now you want to merge all of your branch changes back into the trunk for others to enjoy.

So how do we use **svn merge** in this scenario? Remember that this command compares two trees, and applies the differences to a working copy. So to receive the changes, you need to have a working copy of the trunk. We'll assume that either you still have your original one lying around (fully updated), or that you recently checked out a fresh working copy of `/calc/trunk`.

But which two trees should be compared? At first glance, the answer may seem obvious: just compare the latest trunk tree with your latest branch tree. But beware—this assumption is *wrong*, and has burned many a new user! Since **svn merge** operates like **svn diff**, comparing the latest trunk and branch trees will *not* merely describe the set of changes you made to your branch. Such a comparison shows too many changes: it would not only show the addition of your branch changes, but also the *removal* of trunk changes that never happened on your branch.

To express only the changes that happened on your branch, you need to compare the initial state of your branch to its final state. Using **svn log** on your branch, you can see that your branch was created in revision 341. And the final state of your branch is simply a matter of us-

ing the HEAD revision. That means you want to compare revisions 341 and HEAD of your branch directory, and apply those differences to a working copy of the trunk.



A nice way of finding the revision in which a branch was created (the “base” of the branch) is to use the `--stop-on-copy` option to **svn log**. The log subcommand will normally show every change ever made to the branch, including tracing back through the copy which created the branch. So normally, you'll see history from the trunk as well. The `--stop-on-copy` will halt log output as soon as **svn log** detects that its target was copied or renamed.

So in our continuing example,

```
$ svn log -v --stop-on-copy \
          http://svn.example.com/repos/calc/branches/my-calc-branch
…
------------------------------------------------------------------------
r341 | user | 2002-11-03 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
Changed paths:
   A /calc/branches/my-calc-branch (from /calc/trunk:340)

$
```

As expected, the final revision printed by this command is the revision in which `my-calc-branch` was created by copying.

Here's the final merging procedure, then:

```
$ cd calc/trunk
$ svn update
At revision 405.

$ svn merge -r 341:405 http://svn.example.com/repos/calc/branches/my-calc-branch
U   integer.c
U   button.c
U   Makefile

$ svn status
M   integer.c
M   button.c
M   Makefile

# ...examine the diffs, compile, test, etc...

$ svn commit -m "Merged my-calc-branch changes r341:405 into the trunk."
Sending        integer.c
Sending        button.c
Sending        Makefile
Transmitting file data ...
Committed revision 406.
```

Again, notice that the commit log message very specifically mentions the range of changes that was merged into the trunk. Always remember to do this, because it's critical information you'll need later on.

For example, suppose you decide to keep working on your branch for another week, in order to complete an enhancement to your original feature or bug fix. The repository's HEAD revision

is now 480, and you're ready to do another merge from your private branch to the trunk. But as discussed in the section called “Best Practices for Merging”, you don't want to merge the changes you've already merged before; you only want to merge everything “new” on your branch since the last time you merged. The trick is to figure out what's new.

The first step is to run **svn log** on the trunk, and look for a log message about the last time you merged from the branch:

```
$ cd calc/trunk
$ svn log
…
------------------------------------------------------------------------
r406 | user | 2004-02-08 11:17:26 -0600 (Sun, 08 Feb 2004) | 1 line

Merged my-calc-branch changes r341:405 into the trunk.
------------------------------------------------------------------------
…
```

Aha! Since all branch-changes that happened between revisions 341 and 405 were previously merged to the trunk as revision 406, you now know that you want to merge only the branch changes after that—by comparing revisions 406 and `HEAD`.

```
$ cd calc/trunk
$ svn update
At revision 480.

# We notice that HEAD is currently 480, so we use it to do the merge:

$ svn merge -r 406:480 http://svn.example.com/repos/calc/branches/my-calc-branch
U   integer.c
U   button.c
U   Makefile

$ svn commit -m "Merged my-calc-branch changes r406:480 into the trunk."
Sending        integer.c
Sending        button.c
Sending        Makefile
Transmitting file data ...
Committed revision 481.
```

Now the trunk contains the complete second wave of changes made to the branch. At this point, you can either delete your branch (we'll discuss this later on), or continue working on your branch and repeat this procedure for subsequent merges.

Undoing Changes

Another common use for **svn merge** is to roll back a change that has already been committed. Suppose you're working away happily on a working copy of `/calc/trunk`, and you discover that the change made way back in revision 303, which changed `integer.c`, is completely wrong. It never should have been committed. You can use **svn merge** to “undo” the change in your working copy, and then commit the local modification to the repository. All you need to do is to specify a *reverse* difference. (You can do this by specifying `--revision 303:302`, or by an equivalent `--change -303`.)

```
$ svn merge -c -303 http://svn.example.com/repos/calc/trunk
U  integer.c
```

```
$ svn status
M  integer.c

$ svn diff
…
# verify that the change is removed
…

$ svn commit -m "Undoing change committed in r303."
Sending        integer.c
Transmitting file data .
Committed revision 350.
```

One way to think about a repository revision is as a specific group of changes (some version control systems call these *changesets*). By using the `-r` option, you can ask **svn merge** to apply a changeset, or whole range of changesets, to your working copy. In our case of undoing a change, we're asking **svn merge** to apply changeset #303 to our working copy *backwards*.

Subversion and Changesets

Everyone seems to have a slightly different definition of “changeset”, or at least a different expectation of what it means for a version control system to have “changeset features”. For our purpose, let's say that a changeset is just a collection of changes with a unique name. The changes might include textual edits to file contents, modifications to tree structure, or tweaks to metadata. In more common speak, a changeset is just a patch with a name you can refer to.

In Subversion, a global revision number N names a tree in the repository: it's the way the repository looked after the Nth commit. It's also the name of an implicit changeset: if you compare tree N with tree N-1, you can derive the exact patch that was committed. For this reason, it's easy to think of “revision N” as not just a tree, but a changeset as well. If you use an issue tracker to manage bugs, you can use the revision numbers to refer to particular patches that fix bugs—for example, “this issue was fixed by revision 9238.”. Somebody can then run **svn log -r9238** to read about the exact changeset which fixed the bug, and run **svn diff -c 9238** to see the patch itself. And Subversion's `merge` command also uses revision numbers. You can merge specific changesets from one branch to another by naming them in the merge arguments: **svn merge -r9237:9238** would merge changeset #9238 into your working copy.

Keep in mind that rolling back a change like this is just like any other **svn merge** operation, so you should use **svn status** and **svn diff** to confirm that your work is in the state you want it to be in, and then use **svn commit** to send the final version to the repository. After committing, this particular changeset is no longer reflected in the `HEAD` revision.

Again, you may be thinking: well, that really didn't undo the commit, did it? The change still exists in revision 303. If somebody checks out a version of the `calc` project between revisions 303 and 349, they'll still see the bad change, right?

Yes, that's true. When we talk about “removing” a change, we're really talking about removing it from `HEAD`. The original change still exists in the repository's history. For most situations, this is good enough. Most people are only interested in tracking the `HEAD` of a project anyway. There are special cases, however, where you really might want to destroy all evidence of the commit. (Perhaps somebody accidentally committed a confidential document.) This isn't so easy, it turns out, because Subversion was deliberately designed to never lose information. Revisions are immutable trees which build upon one another. Removing a revision from history

would cause a domino effect, creating chaos in all subsequent revisions and possibly invalidating all working copies. [3]

Resurrecting Deleted Items

The great thing about version control systems is that information is never lost. Even when you delete a file or directory, it may be gone from the HEAD revision, but the object still exists in earlier revisions. One of the most common questions new users ask is, “How do I get my old file or directory back?”.

The first step is to define exactly **which** item you're trying to resurrect. Here's a useful metaphor: you can think of every object in the repository as existing in a sort of two-dimensional coordinate system. The first coordinate is a particular revision tree, and the second coordinate is a path within that tree. So every version of your file or directory can be defined by a specific coordinate pair. (Remember the “peg revision” syntax—foo.c@224 —mentioned back in the section called “Peg and Operative Revisions”.)

First, you might need to use **svn log** to discover the exact coordinate pair you wish to resurrect. A good strategy is to run **svn log --verbose** in a directory which used to contain your deleted item. The `--verbose (-v)` option shows a list of all changed items in each revision; all you need to do is find the revision in which you deleted the file or directory. You can do this visually, or by using another tool to examine the log output (via **grep**, or perhaps via an incremental search in an editor).

```
$ cd parent-dir
$ svn log -v
…
------------------------------------------------------------------------
r808 | joe | 2003-12-26 14:29:40 -0600 (Fri, 26 Dec 2003) | 3 lines
Changed paths:
   D /calc/trunk/real.c
   M /calc/trunk/integer.c

Added fast fourier transform functions to integer.c.
Removed real.c because code now in double.c.
…
```

In the example, we're assuming that you're looking for a deleted file `real.c`. By looking through the logs of a parent directory, you've spotted that this file was deleted in revision 808. Therefore, the last version of the file to exist was in the revision right before that. Conclusion: you want to resurrect the path `/calc/trunk/real.c` from revision 807.

That was the hard part—the research. Now that you know what you want to restore, you have two different choices.

One option is to use **svn merge** to apply revision 808 “in reverse”. (We've already discussed how to undo changes, see the section called “Undoing Changes”.) This would have the effect of re-adding `real.c` as a local modification. The file would be scheduled for addition, and after a commit, the file would again exist in HEAD.

In this particular example, however, this is probably not the best strategy. Reverse-applying revision 808 would not only schedule `real.c` for addition, but the log message indicates that it would also undo certain changes to `integer.c`, which you don't want. Certainly, you could reverse-merge revision 808 and then **svn revert** the local modifications to `integer.c`, but this

[3] The Subversion project has plans, however, to someday implement a command that would accomplish the task of permanently deleting information. In the meantime, see the section called “svndumpfilter” for a possible workaround.

technique doesn't scale well. What if there were 90 files changed in revision 808?

A second, more targeted strategy is not to use **svn merge** at all, but rather the **svn copy** command. Simply copy the exact revision and path “coordinate pair” from the repository to your working copy:

```
$ svn copy -r 807 \
           http://svn.example.com/repos/calc/trunk/real.c ./real.c

$ svn status
A  +   real.c

$ svn commit -m "Resurrected real.c from revision 807, /calc/trunk/real.c."
Adding         real.c
Transmitting file data .
Committed revision 1390.
```

The plus sign in the status output indicates that the item isn't merely scheduled for addition, but scheduled for addition “with history”. Subversion remembers where it was copied from. In the future, running **svn log** on this file will traverse back through the file's resurrection and through all the history it had prior to revision 807. In other words, this new `real.c` isn't really new; it's a direct descendant of the original, deleted file.

Although our example shows us resurrecting a file, note that these same techniques work just as well for resurrecting deleted directories.

Common Branching Patterns

Version control is most often used for software development, so here's a quick peek at two of the most common branching/merging patterns used by teams of programmers. If you're not using Subversion for software development, feel free to skip this section. If you're a software developer using version control for the first time, pay close attention, as these patterns are often considered best practices by experienced folk. These processes aren't specific to Subversion; they're applicable to any version control system. Still, it may help to see them described in Subversion terms.

Release Branches

Most software has a typical lifecycle: code, test, release, repeat. There are two problems with this process. First, developers need to keep writing new features while quality-assurance teams take time to test supposedly-stable versions of the software. New work cannot halt while the software is tested. Second, the team almost always needs to support older, released versions of software; if a bug is discovered in the latest code, it most likely exists in released versions as well, and customers will want to get that bugfix without having to wait for a major new release.

Here's where version control can help. The typical procedure looks like this:

- *Developers commit all new work to the trunk.* Day-to-day changes are committed to `/trunk`: new features, bugfixes, and so on.
- *The trunk is copied to a “release” branch.* When the team thinks the software is ready for release (say, a 1.0 release), then `/trunk` might be copied to `/branches/1.0`.
- *Teams continue to work in parallel.* One team begins rigorous testing of the release branch, while another team continues new work (say, for version 2.0) on `/trunk`. If bugs are dis-

covered in either location, fixes are ported back and forth as necessary. At some point, however, even that process stops. The branch is “frozen” for final testing right before a release.

- *The branch is tagged and released.* When testing is complete, `/branches/1.0` is copied to `/tags/1.0.0` as a reference snapshot. The tag is packaged and released to customers.

- *The branch is maintained over time.* While work continues on `/trunk` for version 2.0, bugfixes continue to be ported from `/trunk` to `/branches/1.0`. When enough bugfixes have accumulated, management may decide to do a 1.0.1 release: `/branches/1.0` is copied to `/tags/1.0.1`, and the tag is packaged and released.

This entire process repeats as the software matures: when the 2.0 work is complete, a new 2.0 release branch is created, tested, tagged, and eventually released. After some years, the repository ends up with a number of release branches in “maintenance” mode, and a number of tags representing final shipped versions.

Feature Branches

A *feature branch* is the sort of branch that's been the dominant example in this chapter, the one you've been working on while Sally continues to work on `/trunk`. It's a temporary branch created to work on a complex change without interfering with the stability of `/trunk`. Unlike release branches (which may need to be supported forever), feature branches are born, used for a while, merged back to the trunk, then ultimately deleted. They have a finite span of usefulness.

Again, project policies vary widely concerning exactly when it's appropriate to create a feature branch. Some projects never use feature branches at all: commits to `/trunk` are a free-for-all. The advantage to this system is that it's simple—nobody needs to learn about branching or merging. The disadvantage is that the trunk code is often unstable or unusable. Other projects use branches to an extreme: no change is *ever* committed to the trunk directly. Even the most trivial changes are created on a short-lived branch, carefully reviewed and merged to the trunk. Then the branch is deleted. This system guarantees an exceptionally stable and usable trunk at all times, but at the cost of tremendous process overhead.

Most projects take a middle-of-the-road approach. They commonly insist that `/trunk` compile and pass regression tests at all times. A feature branch is only required when a change requires a large number of destabilizing commits. A good rule of thumb is to ask this question: if the developer worked for days in isolation and then committed the large change all at once (so that `/trunk` were never destabilized), would it be too large a change to review? If the answer to that question is “yes”, then the change should be developed on a feature branch. As the developer commits incremental changes to the branch, they can be easily reviewed by peers.

Finally, there's the issue of how to best keep a feature branch in “sync” with the trunk as work progresses. As we mentioned earlier, there's a great risk to working on a branch for weeks or months; trunk changes may continue to pour in, to the point where the two lines of development differ so greatly that it may become a nightmare trying to merge the branch back to the trunk.

This situation is best avoided by regularly merging trunk changes to the branch. Make up a policy: once a week, merge the last week's worth of trunk changes to the branch. Take care when doing this; the merging needs to be hand-tracked to avoid the problem of repeated merges (as described in the section called “Tracking Merges Manually”). You'll need to write careful log messages detailing exactly which revision ranges have been merged already (as demonstrated in the section called “Merging a Whole Branch to Another”). It may sound intimidating, but it's actually pretty easy to do.

At some point, you'll be ready to merge the “synchronized” feature branch back to the trunk. To do this, begin by doing a final merge of the latest trunk changes to the branch. When that's done, the latest versions of branch and trunk will be absolutely identical except for your branch changes. So in this special case, you would merge by comparing the branch with the trunk:

```
$ cd trunk-working-copy

$ svn update
At revision 1910.

$ svn merge http://svn.example.com/repos/calc/trunk@1910 \
            http://svn.example.com/repos/calc/branches/mybranch@1910
U  real.c
U  integer.c
A  newdirectory
A  newdirectory/newfile
…
```

By comparing the HEAD revision of the trunk with the HEAD revision of the branch, you're defining a delta that describes only the changes you made to the branch; both lines of development already have all of the trunk changes.

Another way of thinking about this pattern is that your weekly sync of trunk to branch is analogous to running **svn update** in a working copy, while the final merge step is analogous to running **svn commit** from a working copy. After all, what else *is* a working copy but a very shallow private branch? It's a branch that's only capable of storing one change at a time.

Traversing Branches

The **svn switch** command transforms an existing working copy to reflect a different branch. While this command isn't strictly necessary for working with branches, it provides a nice shortcut. In our earlier example, after creating your private branch, you checked out a fresh working copy of the new repository directory. Instead, you can simply ask Subversion to change your working copy of `/calc/trunk` to mirror the new branch location:

```
$ cd calc

$ svn info | grep URL
URL: http://svn.example.com/repos/calc/trunk

$ svn switch http://svn.example.com/repos/calc/branches/my-calc-branch
U   integer.c
U   button.c
U   Makefile
Updated to revision 341.

$ svn info | grep URL
URL: http://svn.example.com/repos/calc/branches/my-calc-branch
```

After “switching” to the branch, your working copy is no different than what you would get from doing a fresh checkout of the directory. And it's usually more efficient to use this command, because often branches only differ by a small degree. The server sends only the minimal set of changes necessary to make your working copy reflect the branch directory.

The **svn switch** command also takes a `--revision` (`-r`) option, so you need not always move your working copy to the HEAD of the branch.

Of course, most projects are more complicated than our `calc` example, containing multiple subdirectories. Subversion users often follow a specific algorithm when using branches:

1. Copy the project's entire “trunk” to a new branch directory.

2. Switch only *part* of the trunk working copy to mirror the branch.

In other words, if a user knows that the branch-work only needs to happen on a specific subdirectory, they use **svn switch** to move only that subdirectory to the branch. (Or sometimes users will switch just a single working file to the branch!) That way, they can continue to receive normal “trunk” updates to most of their working copy, but the switched portions will remain immune (unless someone commits a change to their branch). This feature adds a whole new dimension to the concept of a “mixed working copy”—not only can working copies contain a mixture of working revisions, but a mixture of repository locations as well.

If your working copy contains a number of switched subtrees from different repository locations, it continues to function as normal. When you update, you'll receive patches to each subtree as appropriate. When you commit, your local changes will still be applied as a single, atomic change to the repository.

Note that while it's okay for your working copy to reflect a mixture of repository locations, these locations must all be within the *same* repository. Subversion repositories aren't yet able to communicate with one another; that's a feature planned for the future. [4]

Switches and Updates

Have you noticed that the output of **svn switch** and **svn update** look the same? The `switch` command is actually a superset of the update command.

When you run **svn update**, you're asking the repository to compare two trees. The repository does so, and then sends a description of the differences back to the client. The only difference between **svn switch** and **svn update** is that the `update` command always compares two identical paths.

That is, if your working copy is a mirror of `/calc/trunk`, then **svn update** will automatically compare your working copy of `/calc/trunk` to `/calc/trunk` in the `HEAD` revision. If you're switching your working copy to a branch, then **svn switch** will compare your working copy of `/calc/trunk` to some *other* branch-directory in the `HEAD` revision.

In other words, an update moves your working copy through time. A switch moves your working copy through time *and* space.

Because **svn switch** is essentially a variant of **svn update**, it shares the same behaviors; any local modifications in your working copy are preserved when new data arrives from the repository. This allows you to perform all sorts of clever tricks.

For example, suppose you have a working copy of `/calc/trunk` and make a number of changes to it. Then you suddenly realize that you meant to make the changes to a branch instead. No problem! When you **svn switch** your working copy to the branch, the local changes will remain. You can then test and commit them to the branch.

[4] You *can*, however, use **svn switch** with the `--relocate` option if the URL of your server changes and you don't want to abandon an existing working copy. See svn switch for more information and an example.

Tags

Another common version control concept is a *tag*. A tag is just a “snapshot” of a project in time. In Subversion, this idea already seems to be everywhere. Each repository revision is exactly that—a snapshot of the filesystem after each commit.

However, people often want to give more human-friendly names to tags, like `release-1.0`. And they want to make snapshots of smaller subdirectories of the filesystem. After all, it's not so easy to remember that release-1.0 of a piece of software is a particular subdirectory of revision 4822.

Creating a Simple Tag

Once again, **svn copy** comes to the rescue. If you want to create a snapshot of `/calc/trunk` exactly as it looks in the `HEAD` revision, then make a copy of it:

```
$ svn copy http://svn.example.com/repos/calc/trunk \
           http://svn.example.com/repos/calc/tags/release-1.0 \
      -m "Tagging the 1.0 release of the 'calc' project."

Committed revision 351.
```

This example assumes that a `/calc/tags` directory already exists. (If it doesn't, you can create it using **svn mkdir**.) After the copy completes, the new `release-1.0` directory is forever a snapshot of how the project looked in the `HEAD` revision at the time you made the copy. Of course you might want to be more precise about exactly which revision you copy, in case somebody else may have committed changes to the project when you weren't looking. So if you know that revision 350 of `/calc/trunk` is exactly the snapshot you want, you can specify it by passing `-r 350` to the **svn copy** command.

But wait a moment: isn't this tag-creation procedure the same procedure we used to create a branch? Yes, in fact, it is. In Subversion, there's no difference between a tag and a branch. Both are just ordinary directories that are created by copying. Just as with branches, the only reason a copied directory is a “tag” is because *humans* have decided to treat it that way: as long as nobody ever commits to the directory, it forever remains a snapshot. If people start committing to it, it becomes a branch.

If you are administering a repository, there are two approaches you can take to managing tags. The first approach is “hands off”: as a matter of project policy, decide where your tags will live, and make sure all users know how to treat the directories they copy in there. (That is, make sure they know not to commit to them.) The second approach is more paranoid: you can use one of the access-control scripts provided with Subversion to prevent anyone from doing anything but creating new copies in the tags-area (See Chapter 6, *Server Configuration*.) The paranoid approach, however, isn't usually necessary. If a user accidentally commits a change to a tag-directory, you can simply undo the change as discussed in the previous section. This is version control, after all.

Creating a Complex Tag

Sometimes you may want your “snapshot” to be more complicated than a single directory at a single revision.

For example, pretend your project is much larger than our `calc` example: suppose it contains a number of subdirectories and many more files. In the course of your work, you may decide that you need to create a working copy that is designed to have specific features and bug

fixes. You can accomplish this by selectively backdating files or directories to particular revisions (using **svn update -r** liberally), or by switching files and directories to particular branches (making use of **svn switch**). When you're done, your working copy is a hodgepodge of repository locations from different revisions. But after testing, you know it's the precise combination of data you need.

Time to make a snapshot. Copying one URL to another won't work here. In this case, you want to make a snapshot of your exact working copy arrangement and store it in the repository. Luckily, **svn copy** actually has four different uses (which you can read about in Chapter 9, *Subversion Complete Reference*), including the ability to copy a working-copy tree to the repository:

```
$ ls
my-working-copy/

$ svn copy my-working-copy http://svn.example.com/repos/calc/tags/mytag

Committed revision 352.
```

Now there is a new directory in the repository, `/calc/tags/mytag`, which is an exact snapshot of your working copy—mixed revisions, URLs, and all.

Other users have found interesting uses for this feature. Sometimes there are situations where you have a bunch of local changes made to your working copy, and you'd like a collaborator to see them. Instead of running **svn diff** and sending a patch file (which won't capture tree changes, symlink changes or changes in properties), you can instead use **svn copy** to “upload” your working copy to a private area of the repository. Your collaborator can then either check out a verbatim copy of your working copy, or use **svn merge** to receive your exact changes.

While this is a nice method for uploading a quick snapshot of your working copy, note that this is *not* a good way to initially create a branch. Branch creation should be an event onto itself, and this method conflates the creation of a branch with extra changes to files, all within a single revision. This makes it very difficult (later on) to identify a single revision number as a branch point.



Have you ever found yourself making some complex edits (in your `/trunk` working copy) and suddenly realized, “hey, these changes ought to be in their own branch?” A great technique to do this can be summarized in two steps:

```
$ svn copy http://svn.example.com/repos/calc/trunk \
           http://svn.example.com/repos/calc/branches/newbranch
Committed revision 353.

$ svn switch http://svn.example.com/repos/calc/branches/newbranch
At revision 353.
```

The **svn switch** command, like **svn update**, preserves your local edits. At this point, your working copy is now a reflection of the newly created branch, and your next **svn commit** invocation will send your changes there.

Branch Maintenance

You may have noticed by now that Subversion is extremely flexible. Because it implements branches and tags with the same underlying mechanism (directory copies), and because branches and tags appear in normal filesystem space, many people find Subversion intimidating. It's almost *too* flexible. In this section, we'll offer some suggestions for arranging and managing your data over time.

Repository Layout

There are some standard, recommended ways to organize a repository. Most people create a `trunk` directory to hold the “main line” of development, a `branches` directory to contain branch copies, and a `tags` directory to contain tag copies. If a repository holds only one project, then often people create these top-level directories:

```
/trunk
/branches
/tags
```

If a repository contains multiple projects, admins typically index their layout by project (see the section called “Planning Your Repository Organization” to read more about “project roots”):

```
/paint/trunk
/paint/branches
/paint/tags
/calc/trunk
/calc/branches
/calc/tags
```

Of course, you're free to ignore these common layouts. You can create any sort of variation, whatever works best for you or your team. Remember that whatever you choose, it's not a permanent commitment. You can reorganize your repository at any time. Because branches and tags are ordinary directories, the **svn move** command can move or rename them however you wish. Switching from one layout to another is just a matter of issuing a series of server-side moves; if you don't like the way things are organized in the repository, just juggle the directories around.

Remember, though, that while moving directories may be easy to do, you need to be considerate of your users as well. Your juggling can be disorienting to users with existing working copies. If a user has a working copy of a particular repository directory, your **svn move** operation might remove the path from the latest revision. When the user next runs **svn update**, she will be told that her working copy represents a path that no longer exists, and the user will be forced to **svn switch** to the new location.

Data Lifetimes

Another nice feature of Subversion's model is that branches and tags can have finite lifetimes, just like any other versioned item. For example, suppose you eventually finish all your work on your personal branch of the `calc` project. After merging all of your changes back into `/calc/trunk`, there's no need for your private branch directory to stick around anymore:

```
$ svn delete http://svn.example.com/repos/calc/branches/my-calc-branch \
             -m "Removing obsolete branch of calc project."

Committed revision 375.
```

And now your branch is gone. Of course it's not really gone: the directory is simply missing from the HEAD revision, no longer distracting anyone. If you use **svn checkout**, **svn switch**, or **svn list** to examine an earlier revision, you'll still be able to see your old branch.

If browsing your deleted directory isn't enough, you can always bring it back. Resurrecting data is very easy in Subversion. If there's a deleted directory (or file) that you'd like to bring back into HEAD, simply use **svn copy -r** to copy it from the old revision:

```
$ svn copy -r 374 http://svn.example.com/repos/calc/branches/my-calc-branch \
                  http://svn.example.com/repos/calc/branches/my-calc-branch

Committed revision 376.
```

In our example, your personal branch had a relatively short lifetime: you may have created it to fix a bug or implement a new feature. When your task is done, so is the branch. In software development, though, it's also common to have two “main” branches running side-by-side for very long periods. For example, suppose it's time to release a stable version of the `calc` project to the public, and you know it's going to take a couple of months to shake bugs out of the software. You don't want people to add new features to the project, but you don't want to tell all developers to stop programming either. So instead, you create a “stable” branch of the software that won't change much:

```
$ svn copy http://svn.example.com/repos/calc/trunk \
         http://svn.example.com/repos/calc/branches/stable-1.0 \
         -m "Creating stable branch of calc project."

Committed revision 377.
```

And now developers are free to continue adding cutting-edge (or experimental) features to `/calc/trunk`, and you can declare a project policy that only bug fixes are to be committed to `/calc/branches/stable-1.0`. That is, as people continue to work on the trunk, a human selectively ports bug fixes over to the stable branch. Even after the stable branch has shipped, you'll probably continue to maintain the branch for a long time—that is, as long as you continue to support that release for customers.

Vendor branches

As is especially the case when developing software, the data that you maintain under version control is often closely related to, or perhaps dependent upon, someone else's data. Generally, the needs of your project will dictate that you stay as up-to-date as possible with the data provided by that external entity without sacrificing the stability of your own project. This scenario plays itself out all the time—anywhere that the information generated by one group of people has a direct effect on that which is generated by another group.

For example, software developers might be working on an application which makes use of a third-party library. Subversion has just such a relationship with the Apache Portable Runtime library (see the section called “The Apache Portable Runtime Library”). The Subversion source code depends on the APR library for all its portability needs. In earlier stages of Subversion's development, the project closely tracked APR's changing API, always sticking to the “bleeding edge” of the library's code churn. Now that both APR and Subversion have matured, Subversion attempts to synchronize with APR's library API only at well-tested, stable release points.

Now, if your project depends on someone else's information, there are several ways that you could attempt to synchronize that information with your own. Most painfully, you could issue oral or written instructions to all the contributors of your project, telling them to make sure that they have the specific versions of that third-party information that your project needs. If the third-party information is maintained in a Subversion repository, you could also use Subversion's externals definitions to effectively “pin down” specific versions of that information to some location in your own working copy directory (see the section called “Externals Definitions”).

But sometimes you want to maintain custom modifications to third-party data in your own version control system. Returning to the software development example, programmers might need to make modifications to that third-party library for their own purposes. These modifications might include new functionality or bug fixes, maintained internally only until they become part of an official release of the third-party library. Or the changes might never be relayed back to the library maintainers, existing solely as custom tweaks to make the library further suit the needs of the software developers.

Now you face an interesting situation. Your project could house its custom modifications to the third-party data in some disjointed fashion, such as using patch files or full-fledged alternate versions of files and directories. But these quickly become maintenance headaches, requiring some mechanism by which to apply your custom changes to the third-party data, and necessitating regeneration of those changes with each successive version of the third-party data that you track.

The solution to this problem is to use *vendor branches*. A vendor branch is a directory tree in your own version control system that contains information provided by a third-party entity, or vendor. Each version of the vendor's data that you decide to absorb into your project is called a *vendor drop*.

Vendor branches provide two benefits. First, by storing the currently supported vendor drop in your own version control system, the members of your project never need to question whether they have the right version of the vendor's data. They simply receive that correct version as part of their regular working copy updates. Secondly, because the data lives in your own Subversion repository, you can store your custom changes to it in-place—you have no more need of an automated (or worse, manual) method for swapping in your customizations.

General Vendor Branch Management Procedure

Managing vendor branches generally works like this. You create a top-level directory (such as `/vendor`) to hold the vendor branches. Then you import the third party code into a subdirectory of that top-level directory. You then copy that subdirectory into your main development branch (for example, `/trunk`) at the appropriate location. You always make your local changes in the main development branch. With each new release of the code you are tracking you bring it into the vendor branch and merge the changes into `/trunk`, resolving whatever conflicts occur between your local changes and the upstream changes.

Perhaps an example will help to clarify this algorithm. We'll use a scenario where your development team is creating a calculator program that links against a third-party complex number arithmetic library, libcomplex. We'll begin with the initial creation of the vendor branch, and the import of the first vendor drop. We'll call our vendor branch directory `libcomplex`, and our code drops will go into a subdirectory of our vendor branch called `current`. And since **svn import** creates all the intermediate parent directories it needs, we can actually accomplish both of these steps with a single command.

```
$ svn import /path/to/libcomplex-1.0 \
             http://svn.example.com/repos/vendor/libcomplex/current \
```

```
                -m 'importing initial 1.0 vendor drop'
…
```

We now have the current version of the libcomplex source code in `/vendor/libcomplex/current`. Now, we tag that version (see the section called “Tags”) and then copy it into the main development branch. Our copy will create a new directory called `libcomplex` in our existing `calc` project directory. It is in this copied version of the vendor data that we will make our customizations.

```
$ svn copy http://svn.example.com/repos/vendor/libcomplex/current  \
           http://svn.example.com/repos/vendor/libcomplex/1.0      \
           -m 'tagging libcomplex-1.0'
…
$ svn copy http://svn.example.com/repos/vendor/libcomplex/1.0  \
           http://svn.example.com/repos/calc/libcomplex        \
           -m 'bringing libcomplex-1.0 into the main branch'
…
```

We check out our project's main branch—which now includes a copy of the first vendor drop—and we get to work customizing the libcomplex code. Before we know it, our modified version of libcomplex is now completely integrated into our calculator program. [5]

A few weeks later, the developers of libcomplex release a new version of their library—version 1.1—which contains some features and functionality that we really want. We'd like to upgrade to this new version, but without losing the customizations we made to the existing version. What we essentially would like to do is to replace our current baseline version of libcomplex 1.0 with a copy of libcomplex 1.1, and then re-apply the custom modifications we previously made to that library to the new version. But we actually approach the problem from the other direction, applying the changes made to libcomplex between versions 1.0 and 1.1 to our modified copy of it.

To perform this upgrade, we check out a copy of our vendor branch, and replace the code in the `current` directory with the new libcomplex 1.1 source code. We quite literally copy new files on top of existing files, perhaps exploding the libcomplex 1.1 release tarball atop our existing files and directories. The goal here is to make our `current` directory contain only the libcomplex 1.1 code, and to ensure that all that code is under version control. Oh, and we want to do this with as little version control history disturbance as possible.

After replacing the 1.0 code with 1.1 code, **svn status** will show files with local modifications as well as, perhaps, some unversioned or missing files. If we did what we were supposed to do, the unversioned files are only those new files introduced in the 1.1 release of libcomplex—we run **svn add** on those to get them under version control. The missing files are files that were in 1.0 but not in 1.1, and on those paths we run **svn delete**. Finally, once our `current` working copy contains only the libcomplex 1.1 code, we commit the changes we made to get it looking that way.

Our `current` branch now contains the new vendor drop. We tag the new version (in the same way we previously tagged the version 1.0 vendor drop), and then merge the differences between the tag of the previous version and the new current version into our main development branch.

```
$ cd working-copies/calc
$ svn merge http://svn.example.com/repos/vendor/libcomplex/1.0      \
```

[5] And entirely bug-free, of course!

```
            http://svn.example.com/repos/vendor/libcomplex/current  \
            libcomplex
… # resolve all the conflicts between their changes and our changes
$ svn commit -m 'merging libcomplex-1.1 into the main branch'
…
```

In the trivial use case, the new version of our third-party tool would look, from a files-and-directories point of view, just like the previous version. None of the libcomplex source files would have been deleted, renamed or moved to different locations—the new version would contain only textual modifications against the previous one. In a perfect world, our modifications would apply cleanly to the new version of the library, with absolutely no complications or conflicts.

But things aren't always that simple, and in fact it is quite common for source files to get moved around between releases of software. This complicates the process of ensuring that our modifications are still valid for the new version of code, and can quickly degrade into a situation where we have to manually recreate our customizations in the new version. Once Subversion knows about the history of a given source file—including all its previous locations—the process of merging in the new version of the library is pretty simple. But we are responsible for telling Subversion how the source file layout changed from vendor drop to vendor drop.

svn_load_dirs.pl

Vendor drops that contain more than a few deletes, additions and moves complicate the process of upgrading to each successive version of the third-party data. So Subversion supplies the **svn_load_dirs.pl** script to assist with this process. This script automates the importing steps we mentioned in the general vendor branch management procedure to make sure that mistakes are minimized. You will still be responsible for using the merge commands to merge the new versions of the third-party data into your main development branch, but **svn_load_dirs.pl** can help you more quickly and easily arrive at that stage.

In short, **svn_load_dirs.pl** is an enhancement to **svn import** that has several important characteristics:

- It can be run at any point in time to bring an existing directory in the repository to exactly match an external directory, performing all the necessary adds and deletes, and optionally performing moves, too.
- It takes care of complicated series of operations between which Subversion requires an intermediate commit—such as before renaming a file or directory twice.
- It will optionally tag the newly imported directory.
- It will optionally add arbitrary properties to files and directories that match a regular expression.

svn_load_dirs.pl takes three mandatory arguments. The first argument is the URL to the base Subversion directory to work in. This argument is followed by the URL—relative to the first argument—into which the current vendor drop will be imported. Finally, the third argument is the local directory to import. Using our previous example, a typical run of **svn_load_dirs.pl** might look like:

```
$ svn_load_dirs.pl http://svn.example.com/repos/vendor/libcomplex \
                   current                                        \
```

```
                    /path/to/libcomplex-1.1
…
```

You can indicate that you'd like **svn_load_dirs.pl** to tag the new vendor drop by passing the `-t` command-line option and specifying a tag name. This tag is another URL relative to the first program argument.

```
$ svn_load_dirs.pl -t libcomplex-1.1                                  \
                   http://svn.example.com/repos/vendor/libcomplex \
                   current                                            \
                   /path/to/libcomplex-1.1
…
```

When you run **svn_load_dirs.pl**, it examines the contents of your existing “current” vendor drop, and compares them with the proposed new vendor drop. In the trivial case, there will be no files that are in one version and not the other, and the script will perform the new import without incident. If, however, there are discrepancies in the file layouts between versions, **svn_load_dirs.pl** will ask you how to resolve those differences. For example, you will have the opportunity to tell the script that you know that the file `math.c` in version 1.0 of libcomplex was renamed to `arithmetic.c` in libcomplex 1.1. Any discrepancies not explained by moves are treated as regular additions and deletions.

The script also accepts a separate configuration file for setting properties on files and directories matching a regular expression that are *added* to the repository. This configuration file is specified to **svn_load_dirs.pl** using the `-p` command-line option. Each line of the configuration file is a whitespace-delimited set of two or four values: a Perl-style regular expression to match the added path against, a control keyword (either `break` or `cont`), and then optionally a property name and value.

```
\.png$              break   svn:mime-type   image/png
\.jpe?g$            break   svn:mime-type   image/jpeg
\.m3u$              cont    svn:mime-type   audio/x-mpegurl
\.m3u$              break   svn:eol-style   LF
.*                  break   svn:eol-style   native
```

For each added path, the configured property changes whose regular expression matches the path are applied in order, unless the control specification is `break` (which means that no more property changes should be applied to that path). If the control specification is `cont`—an abbreviation for `continue`—then matching will continue with the next line of the configuration file.

Any whitespace in the regular expression, property name, or property value must be surrounded by either single or double quote characters. You can escape quote characters that are not used for wrapping whitespace by preceding them with a backslash (\) character. The backslash escapes only quotes when parsing the configuration file, so do not protect any other characters beyond what is necessary for the regular expression.

Summary

We've covered a lot of ground in this chapter. We've discussed the concepts of tags and branches, and demonstrated how Subversion implements these concepts by copying directories with the **svn copy** command. We've shown how to use **svn merge** to copy changes from one branch to another, or roll back bad changes. We've gone over the use of **svn switch** to

create mixed-location working copies. And we've talked about how one might manage the organization and lifetimes of branches in a repository.

Remember the Subversion mantra: branches and tags are cheap. So use them liberally! At the same time, don't forget to use good merging habits. Cheap copies are only useful when you're careful about tracking your merging actions.

Chapter 5. Repository Administration

The Subversion repository is the central storehouse of all your versioned data. As such, it becomes an obvious candidate for all the love and attention an administrator can offer. While the repository is generally a low-maintenance item, it is important to understand how to properly configure and care for it so that potential problems are avoided, and actual problems are safely resolved.

In this chapter, we'll discuss how to create and configure a Subversion repository. We'll also talk about repository maintenance, providing examples of how and when to use the **svnlook** and **svnadmin** tools provided with Subversion. We'll address some common questions and mistakes, and give some suggestions on how to arrange the data in the repository.

If you plan to access a Subversion repository only in the role of a user whose data is under version control (that is, via a Subversion client), you can skip this chapter altogether. However, if you are, or wish to become, a Subversion repository administrator, [1] this chapter is for you.

The Subversion Repository, Defined

Before jumping into the broader topic of repository administration, let's further define what a repository is. How does it look? How does it feel? Does it take its tea hot or iced, sweetened, and with lemon? As an administrator, you'll be expected to understand the composition of a repository both from a literal, OS-level perspective—how a repository looks and acts with respect to non-Subversion tools—and from a logical perspective—dealing with how data is represented *inside* the repository.

Seen through the eyes of a typical file browser application (such as the Windows Explorer) or command-line based filesystem navigation tools, the Subversion repository is just another directory full of stuff. There are some subdirectories with human-readable configuration files in them, some subdirectories with some not-so-human-readable data files, and so on. As in other areas of the Subversion design, modularity is given high regard, and hierarchical organization is preferred to cluttered chaos. So a shallow glance into a typical repository from a nuts-and-bolts perspective is sufficient to reveal the basic components of the repository:

```
$ ls repos
conf/  dav/  db/  format  hooks/  locks/  README.txt
```

Here's a quick fly-by overview of what exactly you're seeing in this directory listing. (Don't get bogged down in the terminology—detailed coverage of these components exists elsewhere in this and other chapters.)

conf
: A directory containing repository configuration files.

dav
: A directory provided to mod_dav_svn for its private housekeeping data.

[1] This may sound really prestigious and lofty, but we're just talking about anyone who is interested in that mysterious realm beyond the working copy where everyone's data hangs out.

db
: The data store for all of your versioned data.

format
: A file that contains a single integer that indicates the version number of the repository layout.

hooks
: A directory full of hook script templates (and hook scripts themselves, once you've installed some).

locks
: A directory for Subversion's repository lock files, used for tracking accessors to the repository.

README.txt
: A file whose contents merely inform its readers that they are looking at a Subversion repository.

Of course, when accessed via the Subversion libraries, this otherwise unremarkable collection of files and directories suddenly becomes an implementation of a virtual, versioned filesystem, complete with customizable event triggers. This filesystem has its own notions of directories and files, very similar to the notions of such things held by real filesystems (such as NTFS, FAT32, ext3, and so on). But this is a special filesystem—it hangs these directories and files from revisions, keeping all the changes you've ever made to them safely stored and forever accessible. This is where the entirety of your versioned data lives.

Strategies for Repository Deployment

Due largely to the simplicity of the overall design of the Subversion repository and the technologies on which it relies, creating and configuring a repository are fairly straightforward tasks. There are a few preliminary decisions you'll want to make, but the actual work involved in any given setup of a Subversion repository is pretty straightforward, tending towards mindless repetition if you find yourself setting up multiples of these things.

Some things you'll want to consider up front, though, are:

- What data do you expect to live in your repository (or repositories), and how will that data be organized?
- Where will your repository live, and how will it be accessed?
- What types of access control and repository event reporting do you need?
- Which of the available types of data store do you want to use?

In this section, we'll try to help you answer those questions.

Planning Your Repository Organization

While Subversion allows you to move around versioned files and directories without any loss of information, and even provides ways of moving whole sets of versioned history from one repository to another, doing so can greatly disrupt the workflow of those who access the reposit-

ory often and come to expect things to be at certain locations. So before creating a new repository, try to peer into the future a bit; plan ahead before placing your data under version control. By conscientiously “laying out” your repository or repositories and their versioned contents ahead of time, you can prevent many future headaches.

Let's assume that as repository administrator, you will be responsible for supporting the version control system for several projects. Your first decision is whether to use a single repository for multiple projects, or to give each project its own repository, or some compromise of these two.

There are benefits to using a single repository for multiple projects, most obviously the lack of duplicated maintenance. A single repository means that there is one set of hook programs, one thing to routinely backup, one thing to dump and load if Subversion releases an incompatible new version, and so on. Also, you can move data between projects easily, and without losing any historical versioning information.

The downside of using a single repository is that different projects may have different requirements in terms of the repository event triggers, such as needing to send commit notification emails to different mailing lists, or having different definitions about what does and does not constitute a legitimate commit. These aren't insurmountable problems, of course—it just means that all of your hook scripts have to be sensitive to the layout of your repository rather than assuming that the whole repository is associated with a single group of people. Also, remember that Subversion uses repository-global revision numbers. While those numbers don't have any particular magical powers, some folks still don't like the fact that even though no changes have been made to their project lately, the youngest revision number for the repository keeps climbing because other projects are actively adding new revisions. [2]

A middle-ground approach can be taken, too. For example, projects can be grouped by how well they relate to each other. You might have a few repositories with a handful of projects in each repository. That way, projects that are likely to want to share data can do so easily, and as new revisions are added to the repository, at least the developers know that those new revisions are at least remotely related to everyone who uses that repository.

After deciding how to organize your projects with respect to repositories, you'll probably want to think about directory hierarchies within the repositories themselves. Because Subversion uses regular directory copies for branching and tagging (see Chapter 4, *Branching and Merging*), the Subversion community recommends that you choose a repository location for each *project root*—the “top-most” directory which contains data related to that project—and then create three subdirectories beneath that root: `trunk`, meaning the directory under which the main project development occurs; `branches`, which is a directory in which to create various named branches of the main development line; `tags`, which is a collection of tree snapshots that are created, and perhaps destroyed, but never changed. [3]

For example, your repository might look like:

```
/
   calc/
      trunk/
      tags/
      branches/
   calendar/
      trunk/
      tags/
```

[2] Whether founded in ignorance or in poorly considered concepts about how to derive legitimate software development metrics, global revision numbers are a silly thing to fear, and *not* the kind of thing you should weigh when deciding how to arrange your projects and repositories.

[3] The `trunk`, `tags`, and `branches` trio are sometimes referred to as “the TTB directories”.

```
      branches/
   spreadsheet/
      trunk/
      tags/
      branches/
   …
```

Note that it doesn't matter where in your repository each project root is. If you have only one project per repository, the logical place to put each project root is at the root of that project's respective repository. If you have multiple projects, you might want to arrange them in groups inside the repository, perhaps putting projects with similar goals or shared code in the same subdirectory, or maybe just grouping them alphabetically. Such an arrangement might look like:

```
/
   utils/
      calc/
         trunk/
         tags/
         branches/
      calendar/
         trunk/
         tags/
         branches/
      …
   office/
      spreadsheet/
         trunk/
         tags/
         branches/
      …
```

Lay out your repository in whatever way you see fit. Subversion does not expect or enforce a particular layout—in its eyes, a directory is a directory is a directory. Ultimately, you should choose the repository arrangement that meets the needs of the people who work on the projects that live there.

In the name of full disclosure, though, we'll mention another very common layout. In this layout, the `trunk`, `tags`, and `branches` directories live in the root directory of your repository, and your projects are in subdirectories beneath those, like:

```
/
   trunk/
      calc/
      calendar/
      spreadsheet/
      …
   tags/
      calc/
      calendar/
      spreadsheet/
      …
   branches/
      calc/
      calendar/
      spreadsheet/
      …
```

There's nothing particularly incorrect about such a layout, but it may or may not seem as intuitive for your users. Especially in large, multi-project situations with many users, those users may tend to be familiar with only one or two of the projects in the repository. But the projects-as-branch-siblings tends to de-emphasize project individuality and focus on the entire set of projects as a single entity. That's a social issue though. We like our originally suggested arrangement for purely practical reasons—it's easier to ask about (or modify, or migrate elsewhere) the entire history of a single project when there's a single repository path that holds the entire history—past, present, tagged, and branched—for that project and that project alone.

Deciding Where and How to Host Your Repository

Before creating your Subversion repository, an obvious question you'll need to answer is where the thing is going to live. This is strongly connected to a myriad of other questions involving how the repository will be accessed (via a Subversion server or directly), by whom (users behind your corporate firewall or the whole world out on the open Internet), what other services you'll be providing around Subversion (repository browsing interfaces, e-mail based commit notification, etc.), your data backup strategy, and so on.

We cover server choice and configuration in Chapter 6, *Server Configuration*, but the point we'd like to briefly make here is simply that the answers to some of these other questions might have implications that force your hand when deciding where your repository will live. For example, certain deployment scenarios might require accessing the repository via a remote filesystem from multiple computers, in which case (as you'll read in the next section) your choice of a repository back-end data store turns out not to be a choice at all because only one of the available back-ends will work in this scenario.

Addressing each possible way to deploy Subversion is both impossible, and outside the scope of this book. We simply encourage you to evaluate your options using these pages and other sources as your reference material, and plan ahead.

Choosing a Data Store

As of version 1.1, Subversion provides two options for the type of underlying data store—often referred to as “the back-end” or, somewhat confusingly, “the (versioned) filesystem”—that each repository uses. One type of data store keeps everything in a Berkeley DB (or BDB) database environment; repositories that use this type are often referred to as being “BDB-backed”. The other type stores data in ordinary flat files, using a custom format. Subversion developers have adopted the habit of referring to this latter data storage mechanism as *FSFS* [4] —a versioned filesystem implementation that uses the native OS filesystem directly—rather than via a database library or some other abstraction layer—to store data.

Table 5.1, “Repository Data Store Comparison” gives a comparative overview of Berkeley DB and FSFS repositories.

Table 5.1. Repository Data Store Comparison

[4] Often pronounced “fuzz-fuzz”, if Jack Repenning has anything to say about it. (This book, however, assumes that the reader is thinking “eff-ess-eff-ess”.)

Category	Feature	Berkeley DB	FSFS
Reliability	Data integrity	when properly deployed, extremely reliable; Berkeley DB 4.4 brings auto-recovery	older versions had some rarely demonstrated, but data-destroying bugs
	Sensitivity to interruptions	very; crashes and permission problems can leave the database “wedged”, requiring journaled recovery procedures	quite insensitive
Accessibility	Usable from a read-only mount	no	yes
	Platform-independent storage	no	yes
	Usable over network filesystems	generally, no	yes
	Group permissions handling	sensitive to user umask problems; best if accessed by only one user	works around umask problems
Scalability	Repository disk usage	larger (especially if logfiles aren't purged)	smaller
	Number of revision trees	database; no problems	some older native filesystems don't scale well with thousands of entries in a single directory
	Directories with many files	slower	faster
Performance	Checking out latest revision	no meaningful difference	no meaningful difference
	Large commits	slower overall, but cost is amortized across the lifetime of the commit	faster overall, but finalization delay may cause client timeouts

There are advantages and disadvantages to each of these two back-end types. Neither of them is more “official” than the other, though the newer FSFS is the default data store as of Subversion 1.2. Both are reliable enough to trust with your versioned data. But as you can see in Table 5.1, “Repository Data Store Comparison”, the FSFS backend provides quite a bit more flexibility in terms of its supported deployment scenarios. More flexibility means you have to work a little harder to find ways to deploy it incorrectly. Those reasons—plus the fact that not using Berkeley DB means there's one fewer component in the system—largely explain why today almost everyone uses the FSFS backend when creating new repositories.

Fortunately, most programs which access Subversion repositories are blissfully ignorant of which back-end data store is in use. And you aren't even necessarily stuck with your first choice of a data store—in the event that you change your mind later, Subversion provides ways of migrating your repository's data into another repository that uses a different back-end data store. We talk more about that later in this chapter.

The following subsections provide a more detailed look at the available data store types.

Berkeley DB

When the initial design phase of Subversion was in progress, the developers decided to use Berkeley DB for a variety of reasons, including its open-source license, transaction support, reliability, performance, API simplicity, thread-safety, support for cursors, and so on.

Berkeley DB provides real transaction support—perhaps its most powerful feature. Multiple processes accessing your Subversion repositories don't have to worry about accidentally clobbering each other's data. The isolation provided by the transaction system is such that for any given operation, the Subversion repository code sees a static view of the database—not a database that is constantly changing at the hand of some other process—and can make decisions based on that view. If the decision made happens to conflict with what another process is doing, the entire operation is rolled back as if it never happened, and Subversion gracefully retries the operation against a new, updated (and yet still static) view of the database.

Another great feature of Berkeley DB is *hot backups*—the ability to backup the database environment without taking it “offline”. We'll discuss how to backup your repository in the section called “Repository Backup”, but the benefits of being able to make fully functional copies of your repositories without any downtime should be obvious.

Berkeley DB is also a very reliable database system when properly used. Subversion uses Berkeley DB's logging facilities, which means that the database first writes to on-disk log files a description of any modifications it is about to make, and then makes the modification itself. This is to ensure that if anything goes wrong, the database system can back up to a previous *checkpoint*—a location in the log files known not to be corrupt—and replay transactions until the data is restored to a usable state. See the section called “Managing Disk Space” for more about Berkeley DB log files.

But every rose has its thorn, and so we must note some known limitations of Berkeley DB. First, Berkeley DB environments are not portable. You cannot simply copy a Subversion repository that was created on a Unix system onto a Windows system and expect it to work. While much of the Berkeley DB database format is architecture independent, there are other aspects of the environment that are not. Secondly, Subversion uses Berkeley DB in a way that will not operate on Windows 95/98 systems—if you need to house a BDB-backed repository on a Windows machine, stick with Windows 2000 or newer.

While Berkeley DB promises to behave correctly on network shares that meet a particular set of specifications, [5] most networked filesystem types and appliances do *not* actually meet those requirements. And in no case can you allow a BDB-backed repository that resides on a network share to be accessed by multiple clients of that share at once (which quite often is the whole point of having the repository live on a network share in the first place).



If you attempt to use Berkeley DB on a non-compliant remote filesystem, the results are unpredictable—you may see mysterious errors right away, or it may be months before you discover that your repository database is subtly corrupted. You should strongly consider using the FSFS data store for repositories that need to live on a network share.

Finally, because Berkeley DB is a library linked directly into Subversion, it's more sensitive to interruptions than a typical relational database system. Most SQL systems, for example, have

[5] Berkeley DB requires that the underlying filesystem implement strict POSIX locking semantics, and more importantly, the ability to map files directly into process memory.

a dedicated server process that mediates all access to tables. If a program accessing the database crashes for some reason, the database daemon notices the lost connection and cleans up any mess left behind. And because the database daemon is the only process accessing the tables, applications don't need to worry about permission conflicts. These things are not the case with Berkeley DB, however. Subversion (and programs using Subversion libraries) access the database tables directly, which means that a program crash can leave the database in a temporarily inconsistent, inaccessible state. When this happens, an administrator needs to ask Berkeley DB to restore to a checkpoint, which is a bit of an annoyance. Other things can cause a repository to “wedge” besides crashed processes, such as programs conflicting over ownership and permissions on the database files.



Berkeley DB 4.4 brings (to Subversion 1.4 and better) the ability for Subversion to automatically and transparently recover Berkeley DB environments in need of such recovery. When a Subversion process attaches to a repository's Berkeley DB environment, it uses some process accounting mechanisms to detect any unclean disconnections by previous processes, performs any necessary recovery, and then continues on as if nothing happened. This doesn't completely eliminate instances of repository wedging, but it does drastically reduce the amount of human interaction required to recover from them.

So while a Berkeley DB repository is quite fast and scalable, it's best used by a single server process running as one user—such as Apache's **httpd** or **svnserve** (see Chapter 6, *Server Configuration*)—rather than accessing it as many different users via `file://` or `svn+ssh://` URLs. If using a Berkeley DB repository directly as multiple users, be sure to read the section called “Supporting Multiple Repository Access Methods”.

FSFS

In mid-2004, a second type of repository storage system—one which doesn't use a database at all—came into being. An FSFS repository stores the changes associated with a revision in a single file, and so all of a repository's revisions can be found in a single subdirectory full of numbered files. Transactions are created in separate subdirectories as individual files. When complete, the transaction file is renamed and moved into the revisions directory, thus guaranteeing that commits are atomic. And because a revision file is permanent and unchanging, the repository also can be backed up while “hot”, just like a BDB-backed repository.

The FSFS revision files describe a revision's directory structure, file contents, and deltas against files in other revision trees. Unlike a Berkeley DB database, this storage format is portable across different operating systems and isn't sensitive to CPU architecture. Because there's no journaling or shared-memory files being used, the repository can be safely accessed over a network filesystem and examined in a read-only environment. The lack of database overhead also means that the overall repository size is a bit smaller.

FSFS has different performance characteristics too. When committing a directory with a huge number of files, FSFS is able to more quickly append directory entries. On the other hand, FSFS writes the latest version of a file as a delta against an earlier version, which means that checking out the latest tree is a bit slower than fetching the fulltexts stored in a Berkeley DB HEAD revision. FSFS also has a longer delay when finalizing a commit, which could in extreme cases cause clients to time out while waiting for a response.

The most important distinction, however, is FSFS's imperviousness to “wedging” when something goes wrong. If a process using a Berkeley DB database runs into a permissions problem or suddenly crashes, the database can be left in an unusable state until an administrator recovers it. If the same scenarios happen to a process using an FSFS repository, the repository isn't affected at all. At worst, some transaction data is left behind.

The only real argument against FSFS is its relative immaturity compared to Berkeley DB. Unlike Berkeley DB, which has years of history, its own dedicated development team and, now, Oracle's mighty name attached to it, [6] FSFS is a much newer bit of engineering. Prior to Subversion 1.4, it was still shaking out some pretty serious data integrity bugs which, while only triggered in very rare cases, nonetheless did occur. That said, FSFS has quickly become the back-end of choice for some of the largest public and private Subversion repositories, and promises a lower barrier to entry for Subversion across the board.

Creating and Configuring Your Repository

In the section called “Strategies for Repository Deployment”, we looked at some of the important decisions that should be made before creating and configuring your Subversion repository. Now, we finally get to get our hands dirty! In this section, we'll see how to actually create a Subversion repository and configure it to perform custom actions when special repository events occur.

Creating the Repository

Subversion repository creation is an incredibly simple task. The **svnadmin** utility that comes with Subversion provides a subcommand (`create`) for doing just that.

```
$ svnadmin create /path/to/repos
```

This creates a new repository in the directory `/path/to/repos`, and with the default filesystem data store. Prior to Subversion 1.2, the default was to use Berkeley DB; the default is now FSFS. You can explicitly choose the filesystem type using the `--fs-type` argument, which accepts as a parameter either `fsfs` or `bdb`.

```
$ # Create an FSFS-backed repository
$ svnadmin create --fs-type fsfs /path/to/repos
$


# Create a Berkeley-DB-backed repository
$ svnadmin create --fs-type bdb /path/to/repos
$
```

After running this simple command, you have a Subversion repository.



The path argument to **svnadmin** is just a regular filesystem path and not a URL like the **svn** client program uses when referring to repositories. Both **svnadmin** and **svnlook** are considered server-side utilities—they are used on the machine where the repository resides to examine or modify aspects of the repository, and are in fact unable to perform tasks across a network. A common mistake made by Subversion newcomers is trying to pass URLs (even “local” `file://` ones) to these two programs.

[6] Oracle bought Sleepycat and its flagship software, Berkeley DB, on Valentine's Day in 2006.

Present in the `db/` subdirectory of your repository is the implementation of the versioned filesystem. Your new repository's versioned filesystem begins life at revision 0, which is defined to consist of nothing but the top-level root (`/`) directory. Initially, revision 0 also has a single revision property, `svn:date`, set to the time at which the repository was created.

Now that you have a repository, it's time to customize it.



While some parts of a Subversion repository—such as the configuration files and hook scripts—are meant to be examined and modified manually, you shouldn't (and shouldn't need to) tamper with the other parts of the repository “by hand”. The **svnadmin** tool should be sufficient for any changes necessary to your repository, or you can look to third-party tools (such as Berkeley DB's tool suite) for tweaking relevant subsections of the repository. Do *not* attempt manual manipulation of your version control history by poking and prodding around in your repository's data store files!

Implementing Repository Hooks

A *hook* is a program triggered by some repository event, such as the creation of a new revision or the modification of an unversioned property. Some hooks (the so-called “pre hooks”) run in advance of a repository operation and provide a means by which to both report what is about to happen and to prevent it from happening at all. Other hooks (the “post hooks”) run after the completion of a repository event, and are useful for performing tasks that examine—but don't modify—the repository. Each hook is handed enough information to tell what that event is (or was), the specific repository changes proposed (or completed), and the username of the person who triggered the event.

The `hooks` subdirectory is, by default, filled with templates for various repository hooks.

```
$ ls repos/hooks/
post-commit.tmpl   post-unlock.tmpl  pre-revprop-change.tmpl
post-lock.tmpl          pre-commit.tmpl   pre-unlock.tmpl
post-revprop-change.tmpl  pre-lock.tmpl     start-commit.tmpl
```

There is one template for each hook that the Subversion repository supports, and by examining the contents of those template scripts, you can see what triggers each script to run and what data is passed to that script. Also present in many of these templates are examples of how one might use that script, in conjunction with other Subversion-supplied programs, to perform common useful tasks. To actually install a working hook, you need only place some executable program or script into the `repos/hooks` directory which can be executed as the name (like **start-commit** or **post-commit**) of the hook.

On Unix platforms, this means supplying a script or program (which could be a shell script, a Python program, a compiled C binary, or any number of other things) named exactly like the name of the hook. Of course, the template files are present for more than just informational purposes—the easiest way to install a hook on Unix platforms is to simply copy the appropriate template file to a new file that lacks the `.tmpl` extension, customize the hook's contents, and ensure that the script is executable. Windows, however, uses file extensions to determine whether or not a program is executable, so you would need to supply a program whose basename is the name of the hook, and whose extension is one of the special extensions recognized by Windows for executable programs, such as `.exe` for programs, and `.bat` for batch files.



For security reasons, the Subversion repository executes hook programs with an empty environment—that is, no environment variables are set at all, not even `$PATH` (or `%PATH%`, under Windows). Because of this, many administrators are baffled when their hook program runs fine by hand, but doesn't work when run by Subversion. Be sure to explicitly set any necessary environment variables in your hook program and/or use absolute paths to programs.

Subversion executes hooks as the same user who owns the process which is accessing the Subversion repository. In most cases, the repository is being accessed via a Subversion server, so this user is the same user as which that server runs on the system. The hooks themselves will need to be configured with OS-level permissions that allow that user to execute them. Also, this means that any file or programs (including the Subversion repository itself) accessed directly or indirectly by the hook will be accessed as the same user. In other words, be alert to potential permission-related problems that could prevent the hook from performing the tasks it is designed to perform.

There are nine hooks implemented by the Subversion repository, and you can get details about each of them in the section called “Repository Hooks”. As a repository administrator, you'll need to decide which of hooks you wish to implement (by way of providing an appropriately named and permissioned hook program), and how. When you make this decision, keep in mind the big picture of how your repository is deployed. For example, if you are using server configuration to determine which users are permitted to commit changes to your repository, then you don't need to do this sort of access control via the hook system.

There is no shortage of Subversion hook programs and scripts freely available either from the Subversion community itself or elsewhere. These scripts cover a wide range of utility—basic access control, policy adherence checking, issue tracker integration, email- or syndication-based commit notification, and beyond. See Appendix D, *Third Party Tools* for discussion of some of the most commonly used hook programs. Or, if you wish to write your own, see Chapter 8, *Embedding Subversion*.



While hook scripts can do almost anything, there is one dimension in which hook script authors should show restraint: do *not* modify a commit transaction using hook scripts. While it might be tempting to use hook scripts to automatically correct errors or shortcomings or policy violations present in the files being committed, doing so can cause problems. Subversion keeps client-side caches of certain bits of repository data, and if you change a commit transaction in this way, those caches become indetectably stale. This inconsistency can lead to surprising and unexpected behavior. Instead of modifying the transaction, you should simply *validate* the transaction in the `pre-commit` hook and reject the commit if it does not meet the desired requirements. As a bonus, your users will learn the value of careful, compliance-minded work habits.

Berkeley DB Configuration

A Berkeley DB environment is an encapsulation of one or more databases, log files, region files and configuration files. The Berkeley DB environment has its own set of default configuration values for things like the number of database locks allowed to be taken out at any given time, or the maximum size of the journaling log files, etc. Subversion's filesystem logic additionally chooses default values for some of the Berkeley DB configuration options. However, sometimes your particular repository, with its unique collection of data and access patterns, might require a different set of configuration option values.

The producers of Berkeley DB understand that different applications and database environments have different requirements, and so they have provided a mechanism for overriding at runtime many of the configuration values for the Berkeley DB environment: BDB checks for the presence of a file named `DB_CONFIG` in the environment directory (namely, the repository's `db` subdirectory), and parses the options found in that file . Subversion itself creates this file when it creates the rest of the repository. The file initially contains some default options, as well as pointers to the Berkeley DB online documentation so you can read about what those options do. Of course, you are free to add any of the supported Berkeley DB options to your `DB_CONFIG` file. Just be aware that while Subversion never attempts to read or interpret the contents of the file, and makes no direct use of the option settings in it, you'll want to avoid any configuration changes that may cause Berkeley DB to behave in a fashion that is at odds with what Subversion might expect. Also, changes made to `DB_CONFIG` won't take effect until you recover the database environment (using **svnadmin recover**).

Repository Maintenance

Maintaining a Subversion repository can be daunting, mostly due to the complexities inherent in systems which have a database backend. Doing the task well is all about knowing the tools—what they are, when to use them, and how to use them. This section will introduce you to the repository administration tools provided by Subversion, and how to wield them to accomplish tasks such as repository data migration, upgrades, backups and cleanups.

An Administrator's Toolkit

Subversion provides a handful of utilities useful for creating, inspecting, modifying and repairing your repository. Let's look more closely at each of those tools. Afterward, we'll briefly examine some of the utilities included in the Berkeley DB distribution that provide functionality specific to your repository's database backend not otherwise provided by Subversion's own tools.

svnadmin

The **svnadmin** program is the repository administrator's best friend. Besides providing the ability to create Subversion repositories, this program allows you to perform several maintenance operations on those repositories. The syntax of **svnadmin** is similar to that of other Subversion command-line programs:

```
$ svnadmin help
general usage: svnadmin SUBCOMMAND REPOS_PATH  [ARGS & OPTIONS ...]
Type 'svnadmin help <subcommand>' for help on a specific subcommand.
Type 'svnadmin --version' to see the program version and FS modules.

Available subcommands:
   crashtest
   create
   deltify
…
```

We've already mentioned **svnadmin**'s `create` subcommand (see the section called “Creating the Repository”). Most of the others we will cover later in this chapter. And you can consult the section called “**svnadmin**” for a full rundown of subcommands and what each of them offers.

svnlook

svnlook is a tool provided by Subversion for examining the various revisions and *transactions*

(which are revisions in-the-making) in a repository. No part of this program attempts to change the repository. **svnlook** is typically used by the repository hooks for reporting the changes that are about to be committed (in the case of the **pre-commit** hook) or that were just committed (in the case of the **post-commit** hook) to the repository. A repository administrator may use this tool for diagnostic purposes.

svnlook has a straightforward syntax:

```
$ svnlook help
general usage: svnlook SUBCOMMAND REPOS_PATH [ARGS & OPTIONS ...]
Note: any subcommand which takes the '--revision' and '--transaction'
      options will, if invoked without one of those options, act on
      the repository's youngest revision.
Type 'svnlook help <subcommand>' for help on a specific subcommand.
Type 'svnlook --version' to see the program version and FS modules.
…
```

Nearly every one of **svnlook**'s subcommands can operate on either a revision or a transaction tree, printing information about the tree itself, or how it differs from the previous revision of the repository. You use the `--revision (-r)` and `--transaction (-t)` options to specify which revision or transaction, respectively, to examine. In the absence of both the `--revision (-r)` and `--transaction (-t)` options, **svnlook** will examine the youngest (or “HEAD”) revision in the repository. So the following two commands do exactly the same thing when 19 is the youngest revision in the repository located at `/path/to/repos`:

```
$ svnlook info /path/to/repos
$ svnlook info /path/to/repos -r 19
```

The only exception to these rules about subcommands is the **svnlook youngest** subcommand, which takes no options, and simply prints out the repository's youngest revision number.

```
$ svnlook youngest /path/to/repos
19
```



Keep in mind that the only transactions you can browse are uncommitted ones. Most repositories will have no such transactions, because transactions are usually either committed (in which case, you should access them as revision with the `--revision (-r)` option) or aborted and removed.

Output from **svnlook** is designed to be both human- and machine-parsable. Take as an example the output of the `info` subcommand:

```
$ svnlook info /path/to/repos
sally
2002-11-04 09:29:13 -0600 (Mon, 04 Nov 2002)
27
Added the usual
Greek tree.
```

The output of the `info` subcommand is defined as:

1. The author, followed by a newline.

2. The date, followed by a newline.

3. The number of characters in the log message, followed by a newline.

4. The log message itself, followed by a newline.

This output is human-readable, meaning items like the datestamp are displayed using a textual representation instead of something more obscure (such as the number of nanoseconds since the Tasty Freeze guy drove by). But the output is also machine-parsable—because the log message can contain multiple lines and be unbounded in length, **svnlook** provides the length of that message before the message itself. This allows scripts and other wrappers around this command to make intelligent decisions about the log message, such as how much memory to allocate for the message, or at least how many bytes to skip in the event that this output is not the last bit of data in the stream.

svnlook can perform a variety of other queries: displaying subsets of bits of information we've mentioned previously, recursively listing versioned directory trees, reporting which paths were modified in a given revision or transaction, showing textual and property differences made to files and directories, and so on. See the section called “**svnlook**” for a full reference of **svnlook**'s features.

svndumpfilter

While it won't be the most commonly used tool at the administrator's disposal, **svndumpfilter** provides a very particular brand of useful functionality—the ability to quickly and easily modify streams of Subversion repository history data by acting as a path-based filter.

The syntax of **svndumpfilter** is as follows:

```
$ svndumpfilter help
general usage: svndumpfilter SUBCOMMAND [ARGS & OPTIONS ...]
Type "svndumpfilter help <subcommand>" for help on a specific subcommand.
Type 'svndumpfilter --version' to see the program version.

Available subcommands:
   exclude
   include
   help (?, h)
```

There are only two interesting subcommands. They allow you to make the choice between explicit or implicit inclusion of paths in the stream:

`exclude`
: Filter out a set of paths from the dump data stream.

`include`
: Allow only the requested set of paths to pass through the dump data stream.

You can learn more about these subcommands and **svndumpfilter**'s unique purpose in the section called “Filtering Repository History”.

svnsync

The **svnsync** program, which is new to the 1.4 release of Subversion, provides all the functionality required for maintaining a read-only mirror of a Subversion repository. The program really has one job—to transfer one repository's versioned history into another repository. And while there are few ways to do that, its primary strength is that it can operate remotely—the "source" and "sink" [7] repositories may be on different computers from each other and from **svnsync** itself.

As you might expect, **svnsync** has a syntax that looks very much like every other program we've mentioned in this chapter:

```
$ svnsync help
general usage: svnsync SUBCOMMAND DEST_URL  [ARGS & OPTIONS ...]
Type 'svnsync help <subcommand>' for help on a specific subcommand.
Type 'svnsync --version' to see the program version and RA modules.

Available subcommands:
   initialize (init)
   synchronize (sync)
   copy-revprops
   help (?, h)
$
```

We talk more about replication repositories with **svnsync** in the section called "Repository Replication".

Berkeley DB Utilities

If you're using a Berkeley DB repository, then all of your versioned filesystem's structure and data live in a set of database tables within the `db/` subdirectory of your repository. This subdirectory is a regular Berkeley DB environment directory, and can therefore be used in conjunction with any of the Berkeley database tools, typically provided as part of the Berkeley DB distribution.

For day-to-day Subversion use, these tools are unnecessary. Most of the functionality typically needed for Subversion repositories has been duplicated in the **svnadmin** tool. For example, **svnadmin list-unused-dblogs** and **svnadmin list-dblogs** perform a subset of what is provided by the Berkeley **db_archive** command, and **svnadmin recover** reflects the common use cases of the **db_recover** utility.

However, there are still a few Berkeley DB utilities that you might find useful. The **db_dump** and **db_load** programs write and read, respectively, a custom file format which describes the keys and values in a Berkeley DB database. Since Berkeley databases are not portable across machine architectures, this format is a useful way to transfer those databases from machine to machine, irrespective of architecture or operating system. As we describe later in this chapter, you can also use **svnadmin dump** and **svnadmin load** for similar purposes, but **db_dump** and **db_load** can do certain jobs just as well and much faster. They can also be useful if the experienced Berkeley DB hacker needs to do in-place tweaking of the data in a BDB-backed repository for some reason, which is something Subversion's utilities won't allow. Also, the **db_stat** utility can provide useful information about the status of your Berkeley DB environment, including detailed statistics about the locking and storage subsystems.

For more information on the Berkeley DB tool chain, visit the documentation section of the

[7] Or is that, the "sync"?

Berkeley DB section of Oracle's website, located at http://www.oracle.com/technology/documentation/berkeley-db/db/.

Commit Log Message Correction

Sometimes a user will have an error in her log message (a misspelling or some misinformation, perhaps). If the repository is configured (using the `pre-revprop-change` hook; see the section called “Implementing Repository Hooks”) to accept changes to this log message after the commit is finished, then the user can “fix” her log message remotely using the **svn** program's `propset` command (see svn propset). However, because of the potential to lose information forever, Subversion repositories are not, by default, configured to allow changes to unversioned properties—except by an administrator.

If a log message needs to be changed by an administrator, this can be done using **svnadmin setlog**. This command changes the log message (the `svn:log` property) on a given revision of a repository, reading the new value from a provided file.

```
$ echo "Here is the new, correct log message" > newlog.txt
$ svnadmin setlog myrepos newlog.txt -r 388
```

The **svnadmin setlog** command, by default, is still bound by the same protections against modifying unversioned properties as a remote client is—the `pre-` and `post-revprop-change` hooks are still triggered, and therefore must be set up to accept changes of this nature. But an administrator can get around these protections by passing the `--bypass-hooks` option to the **svnadmin setlog** command.



Remember, though, that by bypassing the hooks, you are likely avoiding such things as email notifications of property changes, backup systems which track unversioned property changes, and so on. In other words, be very careful about what you are changing, and how you change it.

Managing Disk Space

While the cost of storage has dropped incredibly in the past few years, disk usage is still a valid concern for administrators seeking to version large amounts of data. Every bit of version history information stored in the live repository needs to be backed up elsewhere, perhaps multiple times as part of rotating backup schedules. It is useful to know what pieces of Subversion's repository data need to remain on the live site, which need to be backed up, and which can be safely removed.

How Subversion saves disk space

To keep the repository small, Subversion uses *deltification* (or, “deltified storage”) within the repository itself. Deltification involves encoding the representation of a chunk of data as a collection of differences against some other chunk of data. If the two pieces of data are very similar, this deltification results in storage savings for the deltified chunk—rather than taking up space equal to the size of the original data, it takes up only enough space to say, “I look just like this other piece of data over here, except for the following couple of changes”. The result is that most of the repository data that tends to be bulky—namely, the contents of versioned files—is stored at a much smaller size than the original “fulltext” representation of that data. And for repositories created with Subversion 1.4 or later, the space savings are even better—now those fulltext representations of file contents are themselves compressed.



Because all of the data that is subject to deltification in a BDB-backed repository is stored in a single Berkeley DB database file, reducing the size of the stored values will not immediately reduce the size of the database file itself. Berkeley DB will, however, keep internal records of unused areas of the database file, and consume those areas first before growing the size of the database file. So while deltification doesn't produce immediate space savings, it can drastically slow future growth of the database.

Removing dead transactions

Though they are uncommon, there are circumstances in which a Subversion commit process might fail, leaving behind in the repository the remnants of the revision-to-be that wasn't—an uncommitted transaction and all the file and directory changes associated with it. This could happen for several reasons: perhaps the client operation was inelegantly terminated by the user, or a network failure occurred in the middle of an operation. Regardless of the reason, dead transactions can happen. They don't do any real harm, other than consuming disk space. A fastidious administrator may nonetheless wish to remove them.

You can use **svnadmin**'s `lstxns` command to list the names of the currently outstanding transactions.

```
$ svnadmin lstxns myrepos
19
3a1
a45
$
```

Each item in the resultant output can then be used with **svnlook** (and its `--transaction (-t)` option) to determine who created the transaction, when it was created, what types of changes were made in the transaction—information that is helpful in determining whether or not the transaction is a safe candidate for removal! If you do indeed want to remove a transaction, its name can be passed to **svnadmin rmtxns**, which will perform the cleanup of the transaction. In fact, the `rmtxns` subcommand can take its input directly from the output of `lstxns`!

```
$ svnadmin rmtxns myrepos `svnadmin lstxns myrepos`
$
```

If you use these two subcommands like this, you should consider making your repository temporarily inaccessible to clients. That way, no one can begin a legitimate transaction before you start your cleanup. Example 5.1, “txn-info.sh (Reporting Outstanding Transactions)” contains a bit of shell-scripting that can quickly generate information about each outstanding transaction in your repository.

Example 5.1. txn-info.sh (Reporting Outstanding Transactions)

```
#!/bin/sh

### Generate informational output for all outstanding transactions in
### a Subversion repository.
```

```
REPOS="${1}"
if [ "x$REPOS" = x ] ; then
  echo "usage: $0 REPOS_PATH"
  exit
fi

for TXN in `svnadmin lstxns ${REPOS}`; do
  echo "---[ Transaction ${TXN} ]-------------------------------------------"
  svnlook info "${REPOS}" -t "${TXN}"
done
```

The output of the script is basically a concatenation of several chunks of **svnlook info** output (see the section called “svnlook”), and will look something like:

```
$ txn-info.sh myrepos
---[ Transaction 19 ]-------------------------------------------
sally
2001-09-04 11:57:19 -0500 (Tue, 04 Sep 2001)
0
---[ Transaction 3a1 ]-------------------------------------------
harry
2001-09-10 16:50:30 -0500 (Mon, 10 Sep 2001)
39
Trying to commit over a faulty network.
---[ Transaction a45 ]-------------------------------------------
sally
2001-09-12 11:09:28 -0500 (Wed, 12 Sep 2001)
0
$
```

A long-abandoned transaction usually represents some sort of failed or interrupted commit. A transaction's datestamp can provide interesting information—for example, how likely is it that an operation begun nine months ago is still active?

In short, transaction cleanup decisions need not be made unwisely. Various sources of information—including Apache's error and access logs, Subversion's operational logs, Subversion revision history, and so on—can be employed in the decision-making process. And of course, an administrator can often simply communicate with a seemingly dead transaction's owner (via email, for example) to verify that the transaction is, in fact, in a zombie state.

Purging unused Berkeley DB logfiles

Until recently, the largest offender of disk space usage with respect to BDB-backed Subversion repositories was the log files in which Berkeley DB performs its pre-writes before modifying the actual database files. These files capture all the actions taken along the route of changing the database from one state to another—while the database files, at any given time, reflect a particular state, the log files contain all the many changes along the way *between* states. Thus, they can grow and accumulate quite rapidly.

Fortunately, beginning with the 4.2 release of Berkeley DB, the database environment has the ability to remove its own unused log files automatically. Any repositories created using an **svnadmin** which is compiled against Berkeley DB version 4.2 or greater will be configured for this automatic log file removal. If you don't want this feature enabled, simply pass the `--bdb-log-keep` option to the **svnadmin create** command. If you forget to do this, or change your mind at a later time, simply edit the `DB_CONFIG` file found in your repository's `db` directory, comment out the line which contains the `set_flags DB_LOG_AUTOREMOVE` directive,

and then run **svnadmin recover** on your repository to force the configuration changes to take effect. See the section called “Berkeley DB Configuration” for more information about database configuration.

Without some sort of automatic log file removal in place, log files will accumulate as you use your repository. This is actually somewhat of a feature of the database system—you should be able to recreate your entire database using nothing but the log files, so these files can be useful for catastrophic database recovery. But typically, you'll want to archive the log files that are no longer in use by Berkeley DB, and then remove them from disk to conserve space. Use the **svnadmin list-unused-dblogs** command to list the unused log files:

```
$ svnadmin list-unused-dblogs /path/to/repos
/path/to/repos/log.0000000031
/path/to/repos/log.0000000032
/path/to/repos/log.0000000033
…
$ rm `svnadmin list-unused-dblogs /path/to/repos`
## disk space reclaimed!
```



BDB-backed repositories whose log files are used as part of a backup or disaster recovery plan should *not* make use of the log file autoremoval feature. Reconstruction of a repository's data from log files can only be accomplished when *all* the log files are available. If some of the log files are removed from disk before the backup system has a chance to copy them elsewhere, the incomplete set of backed-up log files is essentially useless.

Berkeley DB Recovery

As mentioned in the section called “Berkeley DB”, a Berkeley DB repository can sometimes be left in frozen state if not closed properly. When this happens, an administrator needs to rewind the database back into a consistent state. This is unique to BDB-backed repositories, though—if you are using FSFS-backed ones instead, this won't apply to you. And for those of you using Subversion 1.4 with Berkeley DB 4.4 or better, you should find that Subversion has become much more resilient in these types of situations. Still, wedged Berkeley DB repositories do occur, and an administrator needs to know how to safely deal with this circumstance.

In order to protect the data in your repository, Berkeley DB uses a locking mechanism. This mechanism ensures that portions of the database are not simultaneously modified by multiple database accessors, and that each process sees the data in the correct state when that data is being read from the database. When a process needs to change something in the database, it first checks for the existence of a lock on the target data. If the data is not locked, the process locks the data, makes the change it wants to make, and then unlocks the data. Other processes are forced to wait until that lock is removed before they are permitted to continue accessing that section of the database. (This has nothing to do with the locks that you, as a user, can apply to versioned files within the repository; we try to clear up the confusion caused by this terminology collision in The three meanings of “lock”.)

In the course of using your Subversion repository, fatal errors or interruptions can prevent a process from having the chance to remove the locks it has placed in the database. The result is that the back-end database system gets “wedged”. When this happens, any attempts to access the repository hang indefinitely (since each new accessor is waiting for a lock to go away—which isn't going to happen).

If this happens to your repository, don't panic. The Berkeley DB filesystem takes advantage of database transactions and checkpoints and pre-write journaling to ensure that only the most

catastrophic of events [8] can permanently destroy a database environment. A sufficiently paranoid repository administrator will have made off-site backups of the repository data in some fashion, but don't head off to the tape backup storage closet just yet.

Instead, use the following recipe to attempt to “unwedge” your repository:

1. Make sure that there are no processes accessing (or attempting to access) the repository. For networked repositories, this means shutting down the Apache HTTP Server or svnserve daemon, too.

2. Become the user who owns and manages the repository. This is important, as recovering a repository while running as the wrong user can tweak the permissions of the repository's files in such a way that your repository will still be inaccessible even after it is “unwedged”.

3. Run the command **svnadmin recover /path/to/repos**. You should see output like this:

   ```
   Repository lock acquired.
   Please wait; recovering the repository may take some time...

   Recovery completed.
   The latest repos revision is 19.
   ```

   This command may take many minutes to complete.

4. Restart the server process.

This procedure fixes almost every case of repository lock-up. Make sure that you run this command as the user that owns and manages the database, not just as `root`. Part of the recovery process might involve recreating from scratch various database files (shared memory regions, for example). Recovering as `root` will create those files such that they are owned by `root`, which means that even after you restore connectivity to your repository, regular users will be unable to access it.

If the previous procedure, for some reason, does not successfully unwedge your repository, you should do two things. First, move your broken repository directory aside (perhaps by renaming it to something like `repos.BROKEN`) and then restore your latest backup of it. Then, send an email to the Subversion user list (at <`users@subversion.tigris.org`>) describing your problem in detail. Data integrity is an extremely high priority to the Subversion developers.

Migrating Repository Data Elsewhere

A Subversion filesystem has its data spread throughout files in the repository, in a fashion generally understood by (and of interest to) only the Subversion developers themselves. However, circumstances may arise that call for all, or some subset, of that data to be copied or moved into another repository.

Subversion provides such functionality by way of repository dump streams. A repository dump stream (often referred to as a “dumpfile” when stored as a file on disk) is a portable, flat file format that describes the various revisions in your repository—what was changed, by whom, when, and so on. This dump stream is the primary mechanism used to marshal versioned history—in whole or in part, with or without modification—between repositories. And Subversion

[8] E.g.: hard drive + huge electromagnet = disaster.

provides the tools necessary for creating and loading these dump streams—the **svnadmin dump** and **svnadmin load** subcommands, respectively.



While the Subversion repository dump format contains human-readable portions and a familiar structure (it resembles an RFC-822 format, the same type of format used for most email), it is *not* a plaintext file format. It is a binary file format, highly sensitive to meddling. For example, many text editors will corrupt the file by automatically converting line endings.

There are many reasons for dumping and loading Subversion repository data. Early in Subversion's life, the most common reason was due to the evolution of Subversion itself. As Subversion matured, there were times when changes made to the back-end database schema caused compatibility issues with previous versions of the repository, so users had to dump their repository data using the previous version of Subversion, and load it into a freshly created repository with the new version of Subversion. Now, these types of schema changes haven't occurred since Subversion's 1.0 release, and the Subversion developers promise not to force users to dump and load their repositories when upgrading between minor versions (such as from 1.3 to 1.4) of Subversion. But there are still other reasons for dumping and loading, including re-deploying a Berkeley DB repository on a new OS or CPU architecture, switching between the Berkeley DB and FSFS back-ends, or (as we'll cover in the section called “Filtering Repository History”) purging versioned data from repository history.

Whatever your reason for migrating repository history, using the **svnadmin dump** and **svnadmin load** subcommands is straightforward. **svnadmin dump** will output a range of repository revisions that are formatted using Subversion's custom filesystem dump format. The dump format is printed to the standard output stream, while informative messages are printed to the standard error stream. This allows you to redirect the output stream to a file while watching the status output in your terminal window. For example:

```
$ svnlook youngest myrepos
26
$ svnadmin dump myrepos > dumpfile
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
…
* Dumped revision 25.
* Dumped revision 26.
```

At the end of the process, you will have a single file (`dumpfile` in the previous example) that contains all the data stored in your repository in the requested range of revisions. Note that **svnadmin dump** is reading revision trees from the repository just like any other “reader” process would (**svn checkout**, for example), so it's safe to run this command at any time.

The other subcommand in the pair, **svnadmin load**, parses the standard input stream as a Subversion repository dump file, and effectively replays those dumped revisions into the target repository for that operation. It also gives informative feedback, this time using the standard output stream:

```
$ svnadmin load newrepos < dumpfile
<<< Started new txn, based on original revision 1
     * adding path : A ... done.
     * adding path : A/B ... done.
     …
```

```
------- Committed new rev 1 (loaded from original rev 1) >>>

<<< Started new txn, based on original revision 2
     * editing path : A/mu ... done.
     * editing path : A/D/G/rho ... done.

------- Committed new rev 2 (loaded from original rev 2) >>>

…

<<< Started new txn, based on original revision 25
     * editing path : A/D/gamma ... done.

------- Committed new rev 25 (loaded from original rev 25) >>>

<<< Started new txn, based on original revision 26
     * adding path : A/Z/zeta ... done.
     * editing path : A/mu ... done.

------- Committed new rev 26 (loaded from original rev 26) >>>
```

The result of a load is new revisions added to a repository—the same thing you get by making commits against that repository from a regular Subversion client. And just as in a commit, you can use hook programs to perform actions before and after each of the commits made during a load process. By passing the `--use-pre-commit-hook` and `--use-post-commit-hook` options to **svnadmin load**, you can instruct Subversion to execute the pre-commit and post-commit hook programs, respectively, for each loaded revision. You might use these, for example, to ensure that loaded revisions pass through the same validation steps that regular commits pass through. Of course, you should use these options with care—if your post-commit hook sends emails to a mailing list for each new commit, you might not want to spew hundreds or thousands of commit emails in rapid succession at that list! You can read more about the use of hook scripts in the section called “Implementing Repository Hooks”.

Note that because **svnadmin** uses standard input and output streams for the repository dump and load process, people who are feeling especially saucy can try things like this (perhaps even using different versions of **svnadmin** on each side of the pipe):

```
$ svnadmin create newrepos
$ svnadmin dump oldrepos | svnadmin load newrepos
```

By default, the dump file will be quite large—much larger than the repository itself. That's because by default every version of every file is expressed as a full text in the dump file. This is the fastest and simplest behavior, and nice if you're piping the dump data directly into some other process (such as a compression program, filtering program, or into a loading process). But if you're creating a dump file for longer-term storage, you'll likely want to save disk space by using the `--deltas` option. With this option, successive revisions of files will be output as compressed, binary differences—just as file revisions are stored in a repository. This option is slower, but results in a dump file much closer in size to the original repository.

We mentioned previously that **svnadmin dump** outputs a range of revisions. Use the `--revision (-r)` option to specify a single revision to dump, or a range of revisions. If you omit this option, all the existing repository revisions will be dumped.

```
$ svnadmin dump myrepos -r 23 > rev-23.dumpfile
$ svnadmin dump myrepos -r 100:200 > revs-100-200.dumpfile
```

As Subversion dumps each new revision, it outputs only enough information to allow a future loader to re-create that revision based on the previous one. In other words, for any given revision in the dump file, only the items that were changed in that revision will appear in the dump. The only exception to this rule is the first revision that is dumped with the current **svnadmin dump** command.

By default, Subversion will not express the first dumped revision as merely differences to be applied to the previous revision. For one thing, there is no previous revision in the dump file! And secondly, Subversion cannot know the state of the repository into which the dump data will be loaded (if it ever is). To ensure that the output of each execution of **svnadmin dump** is self-sufficient, the first dumped revision is by default a full representation of every directory, file, and property in that revision of the repository.

However, you can change this default behavior. If you add the `--incremental` option when you dump your repository, **svnadmin** will compare the first dumped revision against the previous revision in the repository, the same way it treats every other revision that gets dumped. It will then output the first revision exactly as it does the rest of the revisions in the dump range—mentioning only the changes that occurred in that revision. The benefit of this is that you can create several small dump files that can be loaded in succession, instead of one large one, like so:

```
$ svnadmin dump myrepos -r 0:1000 > dumpfile1
$ svnadmin dump myrepos -r 1001:2000 --incremental > dumpfile2
$ svnadmin dump myrepos -r 2001:3000 --incremental > dumpfile3
```

These dump files could be loaded into a new repository with the following command sequence:

```
$ svnadmin load newrepos < dumpfile1
$ svnadmin load newrepos < dumpfile2
$ svnadmin load newrepos < dumpfile3
```

Another neat trick you can perform with this `--incremental` option involves appending to an existing dump file a new range of dumped revisions. For example, you might have a `post-commit` hook that simply appends the repository dump of the single revision that triggered the hook. Or you might have a script that runs nightly to append dump file data for all the revisions that were added to the repository since the last time the script ran. Used like this, **svnadmin dump** can be one way to back up changes to your repository over time in case of a system crash or some other catastrophic event.

The dump format can also be used to merge the contents of several different repositories into a single repository. By using the `--parent-dir` option of **svnadmin load**, you can specify a new virtual root directory for the load process. That means if you have dump files for three repositories, say `calc-dumpfile`, `cal-dumpfile`, and `ss-dumpfile`, you can first create a new repository to hold them all:

```
$ svnadmin create /path/to/projects
$
```

Then, make new directories in the repository which will encapsulate the contents of each of the three previous repositories:

```
$ svn mkdir -m "Initial project roots" \
      file:///path/to/projects/calc \
      file:///path/to/projects/calendar \
      file:///path/to/projects/spreadsheet
Committed revision 1.
$
```

Lastly, load the individual dump files into their respective locations in the new repository:

```
$ svnadmin load /path/to/projects --parent-dir calc < calc-dumpfile
…
$ svnadmin load /path/to/projects --parent-dir calendar < cal-dumpfile
…
$ svnadmin load /path/to/projects --parent-dir spreadsheet < ss-dumpfile
…
$
```

We'll mention one final way to use the Subversion repository dump format—conversion from a different storage mechanism or version control system altogether. Because the dump file format is, for the most part, human-readable, it should be relatively easy to describe generic sets of changes—each of which should be treated as a new revision—using this file format. In fact, the **cvs2svn** utility (see the section called “Converting a Repository from CVS to Subversion”) uses the dump format to represent the contents of a CVS repository so that those contents can be copied into a Subversion repository.

Filtering Repository History

Since Subversion stores your versioned history using, at the very least, binary differencing algorithms and data compression (optionally in a completely opaque database system), attempting manual tweaks is unwise, if not quite difficult, and at any rate strongly discouraged. And once data has been stored in your repository, Subversion generally doesn't provide an easy way to remove that data. [9] But inevitably, there will be times when you would like to manipulate the history of your repository. You might need to strip out all instances of a file that was accidentally added to the repository (and shouldn't be there for whatever reason). [10] Or, perhaps you have multiple projects sharing a single repository, and you decide to split them up into their own repositories. To accomplish tasks like this, administrators need a more manageable and malleable representation of the data in their repositories—the Subversion repository dump format.

As we described in the section called “Migrating Repository Data Elsewhere”, the Subversion repository dump format is a human-readable representation of the changes that you've made to your versioned data over time. You use the **svnadmin dump** command to generate the dump data, and **svnadmin load** to populate a new repository with it (see the section called “Migrating Repository Data Elsewhere”). The great thing about the human-readability aspect of the dump format is that, if you aren't careless about it, you can manually inspect and modify it. Of course, the downside is that if you have three years' worth of repository activity encapsulated in what is likely to be a very large dump file, it could take you a long, long time to manually inspect and modify it.

That's where **svndumpfilter** becomes useful. This program acts as path-based filter for repos-

[9] That's rather the reason you use version control at all, right?

[10] Conscious, cautious removal of certain bits of versioned data is actually supported by real use-cases. That's why an “obliterate” feature has been one of the most highly requested Subversion features, and one which the Subversion developers hope to soon provide.

itory dump streams. Simply give it either a list of paths you wish to keep, or a list of paths you wish to not keep, then pipe your repository dump data through this filter. The result will be a modified stream of dump data that contains only the versioned paths you (explicitly or implicitly) requested.

Let's look a realistic example of how you might use this program. We discuss elsewhere (see the section called “Planning Your Repository Organization”) the process of deciding how to choose a layout for the data in your repositories—using one repository per project or combining them, arranging stuff within your repository, and so on. But sometimes after new revisions start flying in, you rethink your layout and would like to make some changes. A common change is the decision to move multiple projects which are sharing a single repository into separate repositories for each project.

Our imaginary repository contains three projects: `calc`, `calendar`, and `spreadsheet`. They have been living side-by-side in a layout like this:

```
/
   calc/
      trunk/
      branches/
      tags/
   calendar/
      trunk/
      branches/
      tags/
   spreadsheet/
      trunk/
      branches/
      tags/
```

To get these three projects into their own repositories, we first dump the whole repository:

```
$ svnadmin dump /path/to/repos > repos-dumpfile
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
* Dumped revision 3.
…
$
```

Next, run that dump file through the filter, each time including only one of our top-level directories, and resulting in three new dump files:

```
$ svndumpfilter include calc < repos-dumpfile > calc-dumpfile
…
$ svndumpfilter include calendar < repos-dumpfile > cal-dumpfile
…
$ svndumpfilter include spreadsheet < repos-dumpfile > ss-dumpfile
…
$
```

At this point, you have to make a decision. Each of your dump files will create a valid repository, but will preserve the paths exactly as they were in the original repository. This means that even though you would have a repository solely for your `calc` project, that repository would still have a top-level directory named `calc`. If you want your `trunk`, `tags`, and `branches` directories to live in the root of your repository, you might wish to edit your dump files, tweaking

the `Node-path` and `Node-copyfrom-path` headers to no longer have that first `calc/` path component. Also, you'll want to remove the section of dump data that creates the `calc` directory. It will look something like:

```
Node-path: calc
Node-action: add
Node-kind: dir
Content-length: 0
```



If you do plan on manually editing the dump file to remove a top-level directory, make sure that your editor is not set to automatically convert end-of-line characters to the native format (e.g. \r\n to \n), as the content will then not agree with the metadata. This will render the dump file useless.

All that remains now is to create your three new repositories, and load each dump file into the right repository:

```
$ svnadmin create calc; svnadmin load calc < calc-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path : Makefile ... done.
     * adding path : button.c ... done.
…
$ svnadmin create calendar; svnadmin load calendar < cal-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path : Makefile ... done.
     * adding path : cal.c ... done.
…
$ svnadmin create spreadsheet; svnadmin load spreadsheet < ss-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path : Makefile ... done.
     * adding path : ss.c ... done.
…
$
```

Both of **svndumpfilter**'s subcommands accept options for deciding how to deal with “empty” revisions. If a given revision contained only changes to paths that were filtered out, that now-empty revision could be considered uninteresting or even unwanted. So to give the user control over what to do with those revisions, **svndumpfilter** provides the following command-line options:

`--drop-empty-revs`
: Do not generate empty revisions at all—just omit them.

`--renumber-revs`
: If empty revisions are dropped (using the `--drop-empty-revs` option), change the revision numbers of the remaining revisions so that there are no gaps in the numeric sequence.

`--preserve-revprops`
: If empty revisions are not dropped, preserve the revision properties (log message, author, date, custom properties, etc.) for those empty revisions. Otherwise, empty revisions will only contain the original datestamp, and a generated log message that indicates that this revision was emptied by **svndumpfilter**.

While **svndumpfilter** can be very useful, and a huge timesaver, there are unfortunately a couple of gotchas. First, this utility is overly sensitive to path semantics. Pay attention to whether paths in your dump file are specified with or without leading slashes. You'll want to look at the `Node-path` and `Node-copyfrom-path` headers.

```
…
Node-path: spreadsheet/Makefile
…
```

If the paths have leading slashes, you should include leading slashes in the paths you pass to **svndumpfilter include** and **svndumpfilter exclude** (and if they don't, you shouldn't). Further, if your dump file has an inconsistent usage of leading slashes for some reason, [11] you should probably normalize those paths so they all have, or lack, leading slashes.

Also, copied paths can give you some trouble. Subversion supports copy operations in the repository, where a new path is created by copying some already existing path. It is possible that at some point in the lifetime of your repository, you might have copied a file or directory from some location that **svndumpfilter** is excluding, to a location that it is including. In order to make the dump data self-sufficient, **svndumpfilter** needs to still show the addition of the new path—including the contents of any files created by the copy—and not represent that addition as a copy from a source that won't exist in your filtered dump data stream. But because the Subversion repository dump format only shows what was changed in each revision, the contents of the copy source might not be readily available. If you suspect that you have any copies of this sort in your repository, you might want to rethink your set of included/excluded paths, perhaps including the paths that served as sources of your troublesome copy operations, too.

Finally, **svndumpfilter** takes path filtering quite literally. If you are trying to copy the history of a project rooted at `trunk/my-project` and move it into a repository of its own, you would, of course, use the **svndumpfilter include** command to keep all the changes in and under `trunk/my-project`. But the resulting dump file makes no assumptions about the repository into which you plan to load this data. Specifically, the dump data might begin with the revision which added the `trunk/my-project` directory, but it will *not* contain directives which would create the `trunk` directory itself (because `trunk` doesn't match the include filter). You'll need to make sure that any directories which the new dump stream expect to exist actually do exist in the target repository before trying to load the stream into that repository.

Repository Replication

There are several scenarios in which it is quite handy to have a Subversion repository whose version history is exactly the same as some other repository's. Perhaps the most obvious one is the maintenance of a simple backup repository, used when the primary repository has become inaccessible due to a hardware failure, network outage, or other such annoyance. Other scenarios include deploying mirror repositories to distribute heavy Subversion load across multiple servers, use as a soft-upgrade mechanism, and so on.

As of version 1.4, Subversion provides a program for managing scenarios like these—**svnsync**. **svnsync** works by essentially asking the Subversion server to "replay" revisions, one at a time. It then uses that revision information to mimic a commit of the same to another repository. Neither repository needs to be locally accessible to machine on which **svnsync** is running—its parameters are repository URLs, and it does all its work through Subversion's repository access (RA) interfaces. All it requires is read access to the source repository and read/write access to the destination repository.

[11] While **svnadmin dump** has a consistent leading slash policy—to not include them—other programs which generate dump data might not be so consistent.



When using **svnsync** against a remote source repository, the Subversion server for that repository must be running Subversion version 1.4 or better.

Assuming you already have a source repository that you'd like to mirror, the next thing you need is an empty target repository which will actually serve as that mirror. This target repository can use either of the available filesystem data-store back-ends (see the section called “Choosing a Data Store”), but it must not yet have any version history in it. The protocol via which **svnsync** communicates revision information is highly sensitive to mismatches between the versioned histories contained in the source and target repositories. For this reason, while **svnsync** cannot *demand* that the target repository be read-only, [12] allowing the revision history in the target repository to change by any mechanism other than the mirroring process is a recipe for disaster.



Do *not* modify a mirror repository in such a way as to cause its version history to deviate from that of the repository it mirrors. The only commits and revision property modifications that ever occur on that mirror repository should be those performed by the **svnsync** tool.

Another requirement of the target repository is that the **svnsync** process be allowed to modify certain revision properties. **svnsync** stores its bookkeeping information in special revision properties on revision 0 of the destination repository. Because **svnsync** works within the framework of that repository's hook system, the default state of the repository (which is to disallow revision property changes; see pre-revprop-change) is insufficient. You'll need to explicitly implement the pre-revprop-change hook, and your script must allow **svnsync** to set and change its special properties. With those provisions in place, you are ready to start mirroring repository revisions.



It's a good idea to implement authorization measures which allow your repository replication process to perform its tasks while preventing other users from modifying the contents of your mirror repository at all.

Let's walk through the use of **svnsync** in a somewhat typical mirroring scenario. We'll pepper this discourse with practical recommendations which you are free to disregard if they aren't required by or suitable for your environment.

As a service to the fine developers of our favorite version control system, we will be mirroring the public Subversion source code repository and exposing that mirror publicly on the Internet, hosted on a different machine than the one on which the original Subversion source code repository lives. This remote host has a global configuration which permits anonymous users to read the contents of repositories on the host, but requires users to authenticate in order to modify those repositories. (Please forgive us for glossing over the details of Subversion server configuration for the moment—those are covered thoroughly in Chapter 6, *Server Configuration*.) And for no other reason than that it makes for a more interesting example, we'll be driving the replication process from a third machine, the one which we currently find ourselves using.

First, we'll create the repository which will be our mirror. This and the next couple of steps do require shell access to the machine on which the mirror repository will live. Once the repository is all configured, though, we shouldn't need to touch it directly again.

[12] In fact, it can't truly be read-only, or **svnsync** itself would have a tough time copying revision history into it.

```
$ ssh admin@svn.example.com \
      "svnadmin create /path/to/repositories/svn-mirror"
admin@svn.example.com's password: ********
$
```

At this point, we have our repository, and due to our server's configuration, that repository is now “live” on the Internet. Now, because we don't want anything modifying the repository except our replication process, we need a way to distinguish that process from other would-be committers. To do so, we use a dedicated username for our process. Only commits and revision property modifications performed by the special username `syncuser` will be allowed.

We'll use the repository's hook system both to allow the replication process to do what it needs to do, and to enforce that only it is doing those things. We accomplish this by implementing two of the repository event hooks—pre-revprop-change and start-commit. Our `pre-revprop-change` hook script is found in Example 5.2, “Mirror repository's pre-revprop-change hook script”, and basically verifies that the user attempting the property changes is our `syncuser` user. If so, the change is allowed; otherwise, it is denied.

Example 5.2. Mirror repository's pre-revprop-change hook script

```
#!/bin/sh

USER="$3"

if [ "$USER" = "syncuser" ]; then exit 0; fi

echo "Only the syncuser user may change revision properties" >&2
exit 1
```

That covers revision property changes. Now we need to ensure that only the `syncuser` user is permitted to commit new revisions to the repository. We do this using a `start-commit` hook scripts like the one in Example 5.3, “Mirror repository's start-commit hook script”.

Example 5.3. Mirror repository's start-commit hook script

```
#!/bin/sh

USER="$2"

if [ "$USER" = "syncuser" ]; then exit 0; fi

echo "Only the syncuser user may commit new revisions" >&2
exit 1
```

After installing our hook scripts and ensuring that they are executable by the Subversion server, we're finished with the setup of the mirror repository. Now, we get to actually do the mirroring.

The first thing we need to do with **svnsync** is to register in our target repository the fact that it

will be a mirror of the source repository. We do this using the **svnsync initialize** subcommand. Note that the various **svnsync** subcommands provide several of the same authentication-related options that **svn** does: `--username`, `--password`, `--non-interactive`, `--config-dir`, and `--no-auth-cache`.

```
$ svnsync help init
initialize (init): usage: svnsync initialize DEST_URL SOURCE_URL

Initialize a destination repository for synchronization from
another repository.

The destination URL must point to the root of a repository with
no committed revisions.  The destination repository must allow
revision property changes.

You should not commit to, or make revision property changes in,
the destination repository by any method other than 'svnsync'.
In other words, the destination repository should be a read-only
mirror of the source repository.

Valid options:
  --non-interactive        : do no interactive prompting
  --no-auth-cache          : do not cache authentication tokens
  --username arg           : specify a username ARG
  --password arg           : specify a password ARG
  --config-dir arg         : read user configuration files from directory ARG

$ svnsync initialize http://svn.example.com/svn-mirror \
                     http://svn.collab.net/repos/svn \
                     --username syncuser --password syncpass
Copied properties for revision 0.
$
```

Our target repository will now remember that it is a mirror of the public Subversion source code repository. Notice that we provided a username and password as arguments to **svnsync**—that was required by the pre-revprop-change hook on our mirror repository.



The URLs provided to **svnsync** must point to the root directories of the target and source repositories, respectively. The tool does not handle mirroring of repository subtrees.



The initial release of **svnsync** (in Subversion 1.4) has a small shortcoming—the values given to the `--username` and `--password` command-line options get used for authentication against both the source and destination repositories. Obviously, there's no guarantee that the synchronizing user's credentials are the same in both places. In the event that they are not the same, users trying to run **svnsync** in non-interactive mode (with the `--non-interactive` option) might experience problems.

And now comes the fun part. With a single subcommand, we can tell **svnsync** to copy all the as-yet-unmirrored revisions from the source repository to the target. [13] The **svnsync synchronize** subcommand will peek into the special revision properties previously stored on the

[13] Be forewarned that while it will take only a few seconds for the average reader to parse this paragraph and the sample output which follows it, the actual time required to complete such a mirroring operation is, shall we say, quite a bit longer.

target repository, and determine what repository it is mirroring and that the most recently mirrored revision was revision 0. Then it will query the source repository and determine what the latest revision in that repository is. Finally, it asks the source repository's server to start replaying all the revisions between 0 and that latest revision. As **svnsync** get the resulting response from the source repository's server, it begins forwarding those revisions to the target repository's server as new commits.

```
$ svnsync help synchronize
synchronize (sync): usage: svnsync synchronize DEST_URL

Transfer all pending revisions from source to destination.
…
$ svnsync synchronize http://svn.example.com/svn-mirror \
                      --username syncuser --password syncpass
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Committed revision 3.
Copied properties for revision 3.
…
Committed revision 23406.
Copied properties for revision 23406.
Committed revision 23407.
Copied properties for revision 23407.
Committed revision 23408.
Copied properties for revision 23408.
```

Of particular interest here is that for each mirrored revision, there is first a commit of that revision to the target repository, and then property changes follow. This is because the initial commit is performed by (and attributed to) the user `syncuser`, and datestamped with the time as of that revision's creation. Also, Subversion's underlying repository access interfaces don't provide a mechanism for setting arbitrary revision properties as part of a commit. So **svnsync** follows up with an immediate series of property modifications which copy all the revision properties found for that revision in the source repository into the target repository. This also has the effect of fixing the author and datestamp of the revision to match that of the source repository.

Also noteworthy is that **svnsync** performs careful bookkeeping that allows it to be safely interrupted and restarted without ruining the integrity of the mirrored data. If a network glitch occurs while mirroring a repository, simply repeat the **svnsync synchronize** command and it will happily pick up right where it left off. In fact, as new revisions appear in the source repository, this is exactly what you to do in order to keep your mirror up-to-date.

There is, however, one bit of inelegance in the process. Because Subversion revision properties can be changed at any time throughout the lifetime of the repository, and don't leave an audit trail that indicates when they were changed, replication processes have to pay special attention to them. If you've already mirrored the first 15 revisions of a repository and someone then changes a revision property on revision 12, **svnsync** won't know to go back and patch up its copy of revision 12. You'll need to tell it to do so manually by using (or with some additionally tooling around) the **svnsync copy-revprops** subcommand, which simply re-replicates all the revision properties for a particular revision.

```
$ svnsync help copy-revprops
copy-revprops: usage: svnsync copy-revprops DEST_URL REV

Copy all revision properties for revision REV from source to
destination.
```

```
…
$ svnsync copy-revprops http://svn.example.com/svn-mirror 12 \
                        --username syncuser --password syncpass
Copied properties for revision 12.
$
```

That's repository replication in a nutshell. You'll likely want some automation around such a process. For example, while our example was a pull-and-push setup, you might wish to have your primary repository push changes to one or more blessed mirrors as part of its post-commit and post-revprop-change hook implementations. This would enable the mirror to be up-to-date in as near to realtime as is likely possible.

Also, while it isn't very commonplace to do so, **svnsync** does gracefully mirror repositories in which the user as whom it authenticates only has partial read access. It simply copies only the bits of the repository that it is permitted to see. Obviously such a mirror is not useful as a backup solution.

As far as user interaction with repositories and mirrors goes, it *is* possible to have a single working copy that interacts with both, but you'll have to jump through some hoops to make it happen. First, you need to ensure that both the primary and mirror repositories have the same repository UUID (which is not the case by default). You can set the mirror repository's UUID by loading a dump file stub into it which contains the UUID of the primary repository, like so:

```
$ cat - <<EOF | svnadmin load --force-uuid dest
SVN-fs-dump-format-version: 2

UUID: 65390229-12b7-0310-b90b-f21a5aa7ec8e
EOF
$
```

Now that the two repositories have the same UUID, you can use **svn switch --relocate** to point your working copy to whichever of the repositories you wish to operate against, a process which is described in svn switch. There is a possible danger here, though, in that if the primary and mirror repositories aren't in close synchronization, a working copy up-to-date with, and pointing to, the primary repository will, if relocated to point to an out-of-date mirror, become confused about the apparent sudden loss of revisions it fully expects to be present, and throws errors to that effect. If this occurs, you can relocate your working copy back to the primary repository and then either wait until the mirror repository is up-to-date, or backdate your working copy to a revision you know is present in the sync repository and then retry the relocation.

Finally, be aware that the revision-based replication provided by **svnsync** is only that—replication of revisions. It does not include such things as the hook implementations, repository or server configuration data, uncommitted transactions, or information about user locks on repository paths. Only information carried by the Subversion repository dump file format is available for replication.

Repository Backup

Despite numerous advances in technology since the birth of the modern computer, one thing unfortunately rings true with crystalline clarity—sometimes, things go very, very awry. Power outages, network connectivity dropouts, corrupt RAM and crashed hard drives are but a taste of the evil that Fate is poised to unleash on even the most conscientious administrator. And so we arrive at a very important topic—how to make backup copies of your repository data.

There are two types of backup methods available for Subversion repository administrat-

ors—full and incremental. A full backup of the repository involves squirreling away in one sweeping action all the information required to fully reconstruct that repository in the event of a catastrophe. Usually, it means, quite literally, the duplication of the entire repository directory (which includes either a Berkeley DB or FSFS environment). Incremental backups are lesser things, backups of only the portion of the repository data that has changed since the previous backup.

As far as full backups go, the naive approach might seem like a sane one, but unless you temporarily disable all other access to your repository, simply doing a recursive directory copy runs the risk of generating a faulty backup. In the case of Berkeley DB, the documentation describes a certain order in which database files can be copied that will guarantee a valid backup copy. A similar ordering exists for FSFS data. But you don't have to implement these algorithms yourself, because the Subversion development team has already done so. The **svnadmin hotcopy** command takes care of the minutia involved in making a hot backup of your repository. And its invocation is as trivial as Unix's **cp** or Windows' **copy** operations:

```
$ svnadmin hotcopy /path/to/repos /path/to/repos-backup
```

The resulting backup is a fully functional Subversion repository, able to be dropped in as a replacement for your live repository should something go horribly wrong.

When making copies of a Berkeley DB repository, you can even instruct **svnadmin hotcopy** to purge any unused Berkeley DB logfiles (see the section called “Purging unused Berkeley DB logfiles”) from the original repository upon completion of the copy. Simply provide the `--clean-logs` option on the command-line.

```
$ svnadmin hotcopy --clean-logs /path/to/bdb-repos /path/to/bdb-repos-backup
```

Additional tooling around this command is available, too. The `tools/backup/` directory of the Subversion source distribution holds the **hot-backup.py** script. This script adds a bit of backup management atop **svnadmin hotcopy**, allowing you to keep only the most recent configured number of backups of each repository. It will automatically manage the names of the backed-up repository directories to avoid collisions with previous backups, and will “rotate off” older backups, deleting them so only the most recent ones remain. Even if you also have an incremental backup, you might want to run this program on a regular basis. For example, you might consider using **hot-backup.py** from a program scheduler (such as **cron** on Unix systems) which will cause it to run nightly (or at whatever granularity of Time you deem safe).

Some administrators use a different backup mechanism built around generating and storing repository dump data. We described in the section called “Migrating Repository Data Elsewhere” how to use **svnadmin dump --incremental** to perform an incremental backup of a given revision or range of revisions. And of course, there is a full backup variation of this achieved by omitting the `--incremental` option to that command. There is some value in these methods, in that the format of your backed-up information is flexible—it's not tied to a particular platform, versioned filesystem type, or release of Subversion or Berkeley DB. But that flexibility comes at a cost, namely that restoring that data can take a long time—longer with each new revision committed to your repository. Also, as is the case with so many of the various backup methods, revision property changes made to already-backed-up revisions won't get picked up by a non-overlapping, incremental dump generation. For these reasons, we recommend against relying solely on dump-based backup approaches.

As you can see, each of the various backup types and methods has its advantages and disadvantages. The easiest is by far the full hot backup, which will always result in a perfect working replica of your repository. Should something bad happen to your live repository, you can re-

store from the backup with a simple recursive directory copy. Unfortunately, if you are maintaining multiple backups of your repository, these full copies will each eat up just as much disk space as your live repository. Incremental backups, by contrast, tend to be quicker to generate and smaller to store. But the restoration process can be a pain, often involving applying multiple incremental backups. And other methods have their own peculiarities. Administrators need to find the balance between the cost of making the backup and the cost of restoring it.

The **svnsync** program (see the section called “Repository Replication”) actually provides a rather handy middle-ground approach. If you are regularly synchronizing a read-only mirror with your main repository, then in a pinch, your read-only mirror is probably a good candidate for replacing that main repository if it falls over. The primary disadvantage of this method is that only the versioned repository data gets synchronized—repository configuration files, user-specified repository path locks, and other items which might live in the physical repository directory but not *inside* the repository's virtual versioned filesystem are not handled by svnsync.

In any backup scenario, repository administrators need to be aware of how modifications to unversioned revision properties affect their backups. Since these changes do not themselves generate new revisions, they will not trigger post-commit hooks, and may not even trigger the pre-revprop-change and post-revprop-change hooks. [14] And since you can change revision properties without respect to chronological order—you can change any revision's properties at any time—an incremental backup of the latest few revisions might not catch a property modification to a revision that was included as part of a previous backup.

Generally speaking, only the truly paranoid would need to backup their entire repository, say, every time a commit occurred. However, assuming that a given repository has some other redundancy mechanism in place with relatively fine granularity (like per-commit emails or incremental dumps), a hot backup of the database might be something that a repository administrator would want to include as part of a system-wide nightly backup. It's your data—protect it as much as you'd like.

Often, the best approach to repository backups is a diversified one which leverages combinations of the methods described here. The Subversion developers, for example, back up the Subversion source code repository nightly using **hot-backup.py** and an offsite **rsync** of those full backups; keep multiple archives of all the commit and property change notification emails; and have repository mirrors maintained by various volunteers using **svnsync**. Your solution might be similar, but should be catered to your needs and that delicate balance of convenience with paranoia. And whatever you do, validate your backups from time to time—what good is a spare tire that has a hole in it? While all of this might not save your hardware from the iron fist of Fate, [15] it should certainly help you recover from those trying times.

Summary

By now you should have a basic understanding of how to create, configure, and maintain Subversion repositories. We've introduced you to the various tools that will assist you with this task. Throughout the chapter, we've noted common administration pitfalls, and suggestions for avoiding them.

All that remains is for you to decide what exciting data to store in your repository, and finally, how to make it available over a network. The next chapter is all about networking.

[14] **svnadmin setlog** can be called in a way that bypasses the hook interface altogether.
[15] You know—the collective term for all of her “fickle fingers”.

Chapter 6. Server Configuration

A Subversion repository can be accessed simultaneously by clients running on the same machine on which the repository resides using the `file://` method. But the typical Subversion setup involves a single server machine being accessed from clients on computers all over the office—or, perhaps, all over the world.

This chapter describes how to get your Subversion repository exposed outside its host machine for use by remote clients. We will cover Subversion's currently available server mechanisms, discussing the configuration and use of each. After reading this section, you should be able to decide which networking setup is right for your needs, and understand how to enable such a setup on your host computer.

Overview

Subversion was designed with an abstract network layer. This means that a repository can be programmatically accessed by any sort of server process, and the client “repository access” API allows programmers to write plugins that speak relevant network protocols. In theory, Subversion can use an infinite number of network implementations. In practice, there are only two servers at the time of this writing.

Apache is an extremely popular webserver; using the **mod_dav_svn** module, Apache can access a repository and make it available to clients via the WebDAV/DeltaV protocol, which is an extension of HTTP. Because Apache is an extremely extensible web server, it provides a number of features “for free”, such as encrypted SSL communication, logging, integration with a number of third-party authentication systems, and limited built-in web browsing of repositories.

In the other corner is **svnserve**: a small, lightweight server program that speaks a custom protocol with clients. Because its protocol is explicitly designed for Subversion and is stateful (unlike HTTP), it provides significantly faster network operations—but at the cost of some features as well. It only understands CRAM-MD5 authentication, has no logging, no web-browsing, and no option to encrypt network traffic. It is, however, extremely easy to set up and is often the best option for small teams just starting out with Subversion.

A third option is to use **svnserve** tunneled over an SSH connection. Even though this scenario still uses **svnserve**, it differs quite a bit in features from a traditional **svnserve** deployment. SSH is used to encrypt all communication. SSH is also used exclusively to authenticate, so real system accounts are required on the server host (unlike vanilla **svnserve**, which has its own private user accounts.) Finally, because this setup requires that each user spawn a private, temporary **svnserve** process, it's equivalent (from a permissions point of view) to allowing a group of local users to all access the repository via `file://` URLs. Path-based access control has no meaning, since each user is accessing the repository database files directly.

Here's a quick summary of the three typical server deployments.

Table 6.1. Comparison of Subversion Server Options

Feature	Apache + mod_dav_svn	svnserve	svnserve over SSH
Authentication options	HTTP(S) basic auth, X.509 certificates, LDAP, NTLM, or any	CRAM-MD5	SSH

Feature	Apache + mod_dav_svn	svnserve	svnserve over SSH
	other mechanism available to Apache httpd		
User account options	private 'users' file	private 'users' file	system accounts
Authorization options	read/write access can be granted over whole repository, or specified per-path.	read/write access can be granted over whole repository, or specified per-path.	read/write access only grantable over whole repository
Encryption	via optional SSL	none	SSH tunneled
Logging	full Apache logs of each HTTP request, with optional “high-level” logging of general client operations	no logging	no logging
Interoperability	partially usable by other WebDAV clients	only talks to svn clients	only talks to svn clients
Web viewing	limited built-in support, or via 3rd-party tools such as ViewVC	only via 3rd-party tools such as ViewVC	only via 3rd-party tools such as ViewVC
Speed	somewhat slower	somewhat faster	somewhat faster
Initial setup	somewhat complex	extremely simple	moderately simple

Choosing a Server Configuration

So, which server should you use? Which is best?

Obviously, there's no right answer to that question. Every team has different needs, and the different servers all represent different sets of tradeoffs. The Subversion project itself doesn't endorse one server or another, or consider either server more “official” than another.

Here are some reasons why you might choose one deployment over another, as well as reasons you might *not* choose one.

The svnserve Server

Why you might want to use it:

- Quick and easy to set up.
- Network protocol is stateful and noticeably faster than WebDAV.
- No need to create system accounts on server.
- Password is not passed over the network.

Why you might want to avoid it:

- Network protocol is not encrypted.

- Only one choice of authentication method.
- Password is stored in the clear on the server.
- No logging of any kind, not even errors.

svnserve over SSH

Why you might want to use it:

- Network protocol is stateful and noticeably faster than WebDAV.
- You can take advantage of existing ssh accounts and user infrastructure.
- All network traffic is encrypted.

Why you might want to avoid it:

- Only one choice of authentication method.
- No logging of any kind, not even errors.
- Requires users to be in same system group, or use a shared ssh key.
- If used improperly, can lead to file permissions problems.

The Apache HTTP Server

Why you might want to use it:

- Allows Subversion to use any of the numerous authentication systems already integrated with Apache.
- No need to create system accounts on server.
- Full Apache logging.
- Network traffic can be encrypted via SSL.
- HTTP(S) can usually go through corporate firewalls.
- Built-in repository browsing via web browser.
- Repository can be mounted as a network drive for transparent version control. (See the section called “Autoversioning”.)

Why you might want to avoid it:

- Noticeably slower than svnserve, because HTTP is a stateless protocol and requires more turnarounds.
- Initial setup can be complex.

Recommendations

In general, the authors of this book recommend a vanilla **svnserve** installation for small teams just trying to get started with a Subversion server; it's the simplest to set up, and has the fewest maintenance issues. You can always switch to a more complex server deployment as your needs change.

Here are some general recommendations and tips, based on years of supporting users:

- If you're trying to set up the simplest possible server for your group, then a vanilla **svnserve** installation is the easiest, fastest route. Note, however, that your repository data will be transmitted in the clear over the network. If your deployment is entirely within your company's LAN or VPN, this isn't an issue. If the repository is exposed to the wide-open internet, then you might want to make sure the repository's contents aren't sensitive (e.g. it contains only open-source code.)

- If you need to integrate with existing identity systems (LDAP, Active Directory, NTLM, X.509, etc.), then an Apache-based server is your only real option. Similarly, if you absolutely need server-side logs of either server errors or client activities, then an Apache-based server is required.

- If you've decided to use either Apache or stock **svnserve**, create a single `svn` user on your system and run the server process as that user. Be sure to make the repository directory wholly owned by the `svn` user as well. From a security point of view, this keeps the repository data nicely siloed and protected by operating system filesystem permissions, changeable by only the Subversion server process itself.

- If you have an existing infrastructure heavily based on SSH accounts, and if your users already have system accounts on your server machine, then it makes sense to deploy an svnserve-over-ssh solution. Otherwise, we don't widely recommend this option to the public. It's generally considered safer to have your users access the repository via (imaginary) accounts managed by **svnserve** or Apache, rather than by full-blown system accounts. If your deep desire for encrypted communication still draws you to this option, we recommend using Apache with SSL instead.

- Do *not* be seduced by the simple idea of having all of your users access a repository directly via `file://` URLs. Even if the repository is readily available to everyone via network share, this is a bad idea. It removes any layers of protection between the users and the repository: users can accidentally (or intentionally) corrupt the repository database, it becomes hard to take the repository offline for inspection or upgrade, and it can lead to a mess of file-permissions problems (see the section called “Supporting Multiple Repository Access Methods”.) Note that this is also one of the reasons we warn against accessing repositories via `svn+ssh://` URLs—from a security standpoint, it's effectively the same as local users accessing via `file://`, and can entail all the same problems if the administrator isn't careful.

svnserve, a custom server

The **svnserve** program is a lightweight server, capable of speaking to clients over TCP/IP using a custom, stateful protocol. Clients contact an **svnserve** server by using URLs that begin with the `svn://` or `svn+ssh://` scheme. This section will explain the different ways of running **svnserve**, how clients authenticate themselves to the server, and how to configure appropriate access control to your repositories.

Invoking the Server

There are a few different ways to run the **svnserve** program:

- Run **svnserve** as a standalone daemon, listening for requests.
- Have the Unix **inetd** daemon temporarily spawn **svnserve** whenever a request comes in on a certain port.
- Have SSH invoke a temporary **svnserve** over an encrypted tunnel.
- Run **svnserve** as a Windows service.

svnserve as Daemon

The easiest option is to run **svnserve** as a standalone “daemon” process. Use the `-d` option for this:

```
$ svnserve -d
$              # svnserve is now running, listening on port 3690
```

When running **svnserve** in daemon mode, you can use the `--listen-port=` and `--listen-host=` options to customize the exact port and hostname to “bind” to.

Once we successfully start **svnserve** as above, it makes every repository on your system available to the network. A client needs to specify an *absolute* path in the repository URL. For example, if a repository is located at `/usr/local/repositories/project1`, then a client would reach it via `svn://host.example.com/usr/local/repositories/project1`. To increase security, you can pass the `-r` option to **svnserve**, which restricts it to exporting only repositories below that path. For example:

```
$ svnserve -d -r /usr/local/repositories
…
```

Using the `-r` option effectively modifies the location that the program treats as the root of the remote filesystem space. Clients then use URLs that have that path portion removed from them, leaving much shorter (and much less revealing) URLs:

```
$ svn checkout svn://host.example.com/project1
…
```

svnserve via inetd

If you want **inetd** to launch the process, then you need to pass the `-i` (`--inetd`) option. In the example, we've shown the output from running `svnserve -i` at the command line, but note that isn't how you actually start the daemon; see the paragraphs following the example for how to configure **inetd** to start **svnserve**.

```
$ svnserve -i
( success ( 1 2 ( ANONYMOUS ) ( edit-pipeline ) ) )
```

When invoked with the `--inetd` option, **svnserve** attempts to speak with a Subversion client via *stdin* and *stdout* using a custom protocol. This is the standard behavior for a program being run via **inetd**. The IANA has reserved port 3690 for the Subversion protocol, so on a Unix-like system you can add lines to `/etc/services` like these (if they don't already exist):

```
svn             3690/tcp   # Subversion
svn             3690/udp   # Subversion
```

And if your system is using a classic Unix-like **inetd** daemon, you can add this line to `/etc/inetd.conf`:

```
svn stream tcp nowait svnowner /usr/bin/svnserve svnserve -i
```

Make sure “svnowner” is a user which has appropriate permissions to access your repositories. Now, when a client connection comes into your server on port 3690, **inetd** will spawn an **svnserve** process to service it. Of course, you may also want to add `-r` to the configuration line as well, to restrict which repositories are exported.

svnserve over a Tunnel

A third way to invoke **svnserve** is in “tunnel mode”, with the `-t` option. This mode assumes that a remote-service program such as **RSH** or **SSH** has successfully authenticated a user and is now invoking a private **svnserve** process *as that user*. (Note that you, the user, will rarely, if ever, have reason to invoke **svnserve** with the `-t` at the command line; instead, the **SSH** daemon does so for you.) The **svnserve** program behaves normally (communicating via *stdin* and *stdout*), and assumes that the traffic is being automatically redirected over some sort of tunnel back to the client. When **svnserve** is invoked by a tunnel agent like this, be sure that the authenticated user has full read and write access to the repository database files. It's essentially the same as a local user accessing the repository via `file://` URLs.

This option is described in much more detail in the section called “Tunneling over SSH”.

svnserve as Windows Service

If your Windows system is a descendant of Windows NT (2000, 2003, XP, Vista), then you can run **svnserve** as a standard Windows service. This is typically a much nicer experience than running it as a standalone daemon with the `--daemon (-d)` option. Using daemon-mode requires launching a console, typing a command, and then leaving the console window running indefinitely. A Windows service, however, runs in the background, can start at boot time automatically, and can be started and stopped using the same consistent administration interface as other Windows services.

You'll need to define the new service using the command-line tool **SC.EXE**. Much like the **inetd** configuration line, you must specify an exact invocation of **svnserve** for Windows to run at start-up time:

```
C:\> sc create svn
        binpath= "C:\svn\bin\svnserve.exe --service -r C:\repos"
        displayname= "Subversion Server"
        depend= Tcpip
        start= auto
```

This defines a new Windows service named “svn”, and which executes a particular **svnserve.exe** command when started (in this case, rooted at `C:\repos`.) There are a number of caveats in the prior example, however.

First, notice that the **svnserve.exe** program must always be invoked with the `--service` option. Any other options to **svnserve** must then be specified on the same line, but you cannot add conflicting options such as `--daemon (-d)`, `--tunnel`, or `--inetd (-i)`. Options such as `-r` or `--listen-port` are fine, though. Second, be careful about spaces when invoking the **SC.EXE** command: the `key= value` patterns must have no spaces between `key=` and exactly one space before the `value`. Lastly, be careful about spaces in your command-line to be invoked. If a directory name contains spaces (or other characters that need escaping), place the entire inner value of `binpath` in double-quotes, by escaping them:

```
C:\> sc create svn
        binpath= "\"C:\program files\svn\bin\svnserve.exe\" --service -r C:\repos"
        displayname= "Subversion Server"
        depend= Tcpip
        start= auto
```

Also note that the word `binpath` is misleading—its value is a *command line*, not the path to an executable. That's why you need to surround it with quote marks if it contains embedded spaces.

Once the service is defined, it can stopped, started, or queried using standard GUI tools (the Services administrative control panel), or at the command line as well:

```
C:\> net stop svn
C:\> net start svn
```

The service can also be uninstalled (i.e. undefined) by deleting its definition: `sc delete svn`. Just be sure to stop the service first! The **SC.EXE** program has many other subcommands and options; run `sc /?` to learn more about it.

Built-in authentication and authorization

When a client connects to an **svnserve** process, the following things happen:

- The client selects a specific repository.
- The server processes the repository's `conf/svnserve.conf` file, and begins to enforce any authentication and authorization policies defined therein.
- Depending on the situation and authorization policies,
  - the client may be allowed to make requests anonymously, without ever receiving an authentication challenge, OR
  - the client may be challenged for authentication at any time, OR
  - if operating in “tunnel mode”, the client will declare itself to be already externally authenticated.

At the time of writing, the server only knows how to send a CRAM-MD5 [1] authentication chal-

lenge. In essence, the server sends a small amount of data to the client. The client uses the MD5 hash algorithm to create a fingerprint of the data and password combined, then sends the fingerprint as a response. The server performs the same computation with the stored password to verify that the result is identical. *At no point does the actual password travel over the network.*

It's also possible, of course, for the client to be externally authenticated via a tunnel agent, such as **SSH**. In that case, the server simply examines the user it's running as, and uses it as the authenticated username. For more on this, see the section called “Tunneling over SSH”.

As you've already guessed, a repository's `svnserve.conf` file is the central mechanism for controlling authentication and authorization policies. The file has the same format as other configuration files (see the section called “Runtime Configuration Area”): section names are marked by square brackets (`[` and `]`), comments begin with hashes (#), and each section contains specific variables that can be set (`variable = value`). Let's walk through these files and learn how to use them.

Create a 'users' file and realm

For now, the `[general]` section of the `svnserve.conf` has all the variables you need. Begin by changing the values of those variables: choose a name for a file which will contain your usernames and passwords, and choose an authentication realm:

```
[general]
password-db = userfile
realm = example realm
```

The `realm` is a name that you define. It tells clients which sort of “authentication namespace” they're connecting to; the Subversion client displays it in the authentication prompt, and uses it as a key (along with the server's hostname and port) for caching credentials on disk (see the section called “Client Credentials Caching”). The `password-db` variable points to a separate file that contains a list of usernames and passwords, using the same familiar format. For example:

```
[users]
harry = foopassword
sally = barpassword
```

The value of `password-db` can be an absolute or relative path to the users file. For many admins, it's easy to keep the file right in the `conf/` area of the repository, alongside `svnserve.conf`. On the other hand, it's possible you may want to have two or more repositories share the same users file; in that case, the file should probably live in a more public place. The repositories sharing the users file should also be configured to have the same realm, since the list of users essentially defines an authentication realm. Wherever the file lives, be sure to set the file's read and write permissions appropriately. If you know which user(s) **svnserve** will run as, restrict read access to the user file as necessary.

Set access controls

There are two more variables to set in the `svnserve.conf` file: they determine what unauthenticated (anonymous) and authenticated users are allowed to do. The variables `anon-access` and `auth-access` can be set to the values `none`, `read`, or `write`. Setting the value to `none` prohibits both reading and writing; `read` allows read-only access to the repository,

[1] See RFC 2195.

and `write` allows complete read/write access to the repository. For example:

```
[general]
password-db = userfile
realm = example realm

# anonymous users can only read the repository
anon-access = read

# authenticated users can both read and write
auth-access = write
```

The example settings are, in fact, the default values of the variables, should you forget to define them. If you want to be even more conservative, you can block anonymous access completely:

```
[general]
password-db = userfile
realm = example realm

# anonymous users aren't allowed
anon-access = none

# authenticated users can both read and write
auth-access = write
```

The server process not only understands these “blanket” access controls to the repository, but also finer-grained access restrictions placed on specific files and directories within the repository. To make use of this feature, you need to define a file containing more detailed rules, and then set the `authz-db` variable to point to it:

```
[general]
password-db = userfile
realm = example realm

# Specific access rules for specific locations
authz-db = authzfile
```

The syntax of the `authzfile` file is discussed in detail in the section called “Path-Based Authorization”. Note that the `authz-db` variable isn't mutually exclusive with the `anon-access` and `auth-access` variables; if all the variables are defined at once, then *all* of the rules must be satisfied before access is allowed.

Tunneling over SSH

svnserve's built-in authentication can be very handy, because it avoids the need to create real system accounts. On the other hand, some administrators already have well-established SSH authentication frameworks in place. In these situations, all of the project's users already have system accounts and the ability to “SSH into” the server machine.

It's easy to use SSH in conjunction with **svnserve**. The client simply uses the `svn+ssh://` URL scheme to connect:

```
$ whoami
```

```
harry

$ svn list svn+ssh://host.example.com/repos/project
harry@host.example.com's password:  *****

foo
bar
baz
…
```

In this example, the Subversion client is invoking a local **ssh** process, connecting to `host.example.com`, authenticating as the user `harry`, then spawning a private **svnserve** process on the remote machine running as the user `harry`. The **svnserve** command is being invoked in tunnel mode (`-t`) and its network protocol is being "tunneled" over the encrypted connection by **ssh**, the tunnel-agent. **svnserve** is aware that it's running as the user `harry`, and if the client performs a commit, the authenticated username will be used as the author of the new revision.

The important thing to understand here is that the Subversion client is *not* connecting to a running **svnserve** daemon. This method of access doesn't require a daemon, nor does it notice one if present. It relies wholly on the ability of **ssh** to spawn a temporary **svnserve** process, which then terminates when the network connection is closed.

When using `svn+ssh://` URLs to access a repository, remember that it's the **ssh** program prompting for authentication, and *not* the **svn** client program. That means there's no automatic password caching going on (see the section called "Client Credentials Caching"). The Subversion client often makes multiple connections to the repository, though users don't normally notice this due to the password caching feature. When using `svn+ssh://` URLs, however, users may be annoyed by **ssh** repeatedly asking for a password for every outbound connection. The solution is to use a separate SSH password-caching tool like **ssh-agent** on a Unix-like system, or **pageant** on Windows.

When running over a tunnel, authorization is primarily controlled by operating system permissions to the repository's database files; it's very much the same as if Harry were accessing the repository directly via a `file://` URL. If multiple system users are going to be accessing the repository directly, you may want to place them into a common group, and you'll need to be careful about umasks. (Be sure to read the section called "Supporting Multiple Repository Access Methods".) But even in the case of tunneling, the `svnserve.conf` file can still be used to block access, by simply setting `auth-access = read` or `auth-access = none`. [2]

You'd think that the story of SSH tunneling would end here, but it doesn't. Subversion allows you to create custom tunnel behaviors in your run-time `config` file (see the section called "Runtime Configuration Area"). For example, suppose you want to use RSH instead of SSH[3]. In the `[tunnels]` section of your `config` file, simply define it like this:

```
[tunnels]
rsh = rsh
```

And now, you can use this new tunnel definition by using a URL scheme that matches the name of your new variable: `svn+rsh://host/path`. When using the new URL scheme, the Subversion client will actually be running the command **rsh host svnserve -t** behind the scenes. If you include a username in the URL (for example,

[2] Note that using any sort of **svnserve**-enforced access control at all is a bit pointless; the user already has direct access to the repository database.
[3] We don't actually recommend this, since RSH is notably less secure than SSH.

`svn+rsh://username@host/path`) the client will also include that in its command (**rsh username@host svnserve -t**). But you can define new tunneling schemes to be much more clever than that:

```
[tunnels]
joessh = $JOESSH /opt/alternate/ssh -p 29934
```

This example demonstrates a couple of things. First, it shows how to make the Subversion client launch a very specific tunneling binary (the one located at `/opt/alternate/ssh`) with specific options. In this case, accessing a `svn+joessh://` URL would invoke the particular SSH binary with `-p 29934` as arguments—useful if you want the tunnel program to connect to a non-standard port.

Second, it shows how to define a custom environment variable that can override the name of the tunneling program. Setting the `SVN_SSH` environment variable is a convenient way to override the default SSH tunnel agent. But if you need to have several different overrides for different servers, each perhaps contacting a different port or passing a different set of options to SSH, you can use the mechanism demonstrated in this example. Now if we were to set the `JOESSH` environment variable, its value would override the entire value of the tunnel variable—**$JOESSH** would be executed instead of **/opt/alternate/ssh -p 29934**.

SSH configuration tricks

It's not only possible to control the way in which the client invokes **ssh**, but also to control the behavior of **sshd** on your server machine. In this section, we'll show how to control the exact **svnserve** command executed by **sshd**, as well as how to have multiple users share a single system account.

Initial setup

To begin, locate the home directory of the account you'll be using to launch **svnserve**. Make sure the account has an SSH public/private keypair installed, and that the user can log in via public-key authentication. Password authentication will not work, since all of the following SSH tricks revolve around using the SSH `authorized_keys` file.

If it doesn't already exist, create the `authorized_keys` file (on Unix, typically `~/.ssh/authorized_keys`). Each line in this file describes a public key that is allowed to connect. The lines are typically of the form:

```
ssh-dsa AAAABtce9euch… user@example.com
```

The first field describes the type of key, the second field is the base64-encoded key itself, and the third field is a comment. However, it's a lesser known fact that the entire line can be preceded by a `command` field:

```
command="program" ssh-dsa AAAABtce9euch… user@example.com
```

When the `command` field is set, the SSH daemon will run the named program instead of the typical **svnserve -t** invocation that the Subversion client asks for. This opens the door to a number of server-side tricks. In the following examples, we abbreviate the lines of the file as:

```
command="program" TYPE KEY COMMENT
```

Controlling the invoked command

Because we can specify the executed server-side command, it's easy to name a specific **svnserve** binary to run and to pass it extra arguments:

```
command="/path/to/svnserve -t -r /virtual/root" TYPE KEY COMMENT
```

In this example, `/path/to/svnserve` might be a custom wrapper script around **svnserve** which sets the umask (see the section called “Supporting Multiple Repository Access Methods”). It also shows how to anchor **svnserve** in a virtual root directory, just as one often does when running **svnserve** as a daemon process. This might be done either to restrict access to parts of the system, or simply to relieve the user of having to type an absolute path in the `svn+ssh://` URL.

It's also possible to have multiple users share a single account. Instead of creating a separate system account for each user, generate a public/private keypair for each person. Then place each public key into the `authorized_users` file, one per line, and use the `--tunnel-user` option:

```
command="svnserve -t --tunnel-user=harry" TYPE1 KEY1 harry@example.com
command="svnserve -t --tunnel-user=sally" TYPE2 KEY2 sally@example.com
```

This example allows both Harry and Sally to connect to the same account via public-key authentication. Each of them has a custom command that will be executed; the `--tunnel-user` option tells **svnserve -t** to assume that the named argument is the authenticated user. Without `--tunnel-user`, it would appear as though all commits were coming from the one shared system account.

A final word of caution: giving a user access to the server via public-key in a shared account might still allow other forms of SSH access, even if you've set the `command` value in `authorized_keys`. For example, the user may still get shell access through SSH, or be able to perform X11 or general port-forwarding through your server. To give the user as little permission as possible, you may want to specify a number of restrictive options immediately after the `command`:

```
command="svnserve -t --tunnel-user=harry",no-port-forwarding,\
        no-agent-forwarding,no-X11-forwarding,no-pty \
        TYPE1 KEY1 harry@example.com
```

httpd, the Apache HTTP server

The Apache HTTP Server is a “heavy duty” network server that Subversion can leverage. Via a custom module, **httpd** makes Subversion repositories available to clients via the WebDAV/DeltaV protocol, which is an extension to HTTP 1.1 (see http://www.webdav.org/ for more information). This protocol takes the ubiquitous HTTP protocol that is the core of the World Wide Web, and adds writing—specifically, versioned writing—capabilities. The result is a standardized, robust system that is conveniently packaged as part of the Apache 2.0 software, is supported by numerous operating systems and third-party products, and doesn't require network administrators to open up yet another custom port. [4] While an Apache-Subversion server has

[4] They really hate doing that.

more features than **svnserve**, it's also a bit more difficult to set up. With flexibility often comes more complexity.

Much of the following discussion includes references to Apache configuration directives. While some examples are given of the use of these directives, describing them in full is outside the scope of this chapter. The Apache team maintains excellent documentation, publicly available on their website at http://httpd.apache.org. For example, a general reference for the configuration directives is located at http://httpd.apache.org/docs-2.0/mod/directives.html.

Also, as you make changes to your Apache setup, it is likely that somewhere along the way a mistake will be made. If you are not already familiar with Apache's logging subsystem, you should become aware of it. In your `httpd.conf` file are directives that specify the on-disk locations of the access and error logs generated by Apache (the `CustomLog` and `ErrorLog` directives, respectively). Subversion's mod_dav_svn uses Apache's error logging interface as well. You can always browse the contents of those files for information that might reveal the source of a problem that is not clearly noticeable otherwise.

Why Apache 2?

If you're a system administrator, it's very likely that you're already running the Apache web server and have some prior experience with it. At the time of writing, Apache 1.3 is by far the most popular version of Apache. The world has been somewhat slow to upgrade to the Apache 2.X series for various reasons: some people fear change, especially changing something as critical as a web server. Other people depend on plug-in modules that only work against the Apache 1.3 API, and are waiting for a 2.X port. Whatever the reason, many people begin to worry when they first discover that Subversion's Apache module is written specifically for the Apache 2 API.

The proper response to this problem is: don't worry about it. It's easy to run Apache 1.3 and Apache 2 side-by-side; simply install them to separate places, and use Apache 2 as a dedicated Subversion server that runs on a port other than 80. Clients can access the repository by placing the port number into the URL:

```
$ svn checkout http://host.example.com:7382/repos/project
…
```

Prerequisites

To network your repository over HTTP, you basically need four components, available in two packages. You'll need Apache **httpd** 2.0, the **mod_dav** DAV module that comes with it, Subversion, and the **mod_dav_svn** filesystem provider module distributed with Subversion. Once you have all of those components, the process of networking your repository is as simple as:

- getting httpd 2.0 up and running with the mod_dav module,
- installing the mod_dav_svn plugin to mod_dav, which uses Subversion's libraries to access the repository, and
- configuring your `httpd.conf` file to export (or expose) the repository.

You can accomplish the first two items either by compiling **httpd** and Subversion from source code, or by installing pre-built binary packages of them on your system. For the most up-

to-date information on how to compile Subversion for use with the Apache HTTP Server, as well as how to compile and configure Apache itself for this purpose, see the `INSTALL` file in the top level of the Subversion source code tree.

Basic Apache Configuration

Once you have all the necessary components installed on your system, all that remains is the configuration of Apache via its `httpd.conf` file. Instruct Apache to load the mod_dav_svn module using the `LoadModule` directive. This directive must precede any other Subversion-related configuration items. If your Apache was installed using the default layout, your **mod_dav_svn** module should have been installed in the `modules` subdirectory of the Apache install location (often `/usr/local/apache2`). The `LoadModule` directive has a simple syntax, mapping a named module to the location of a shared library on disk:

```
LoadModule dav_svn_module     modules/mod_dav_svn.so
```

Note that if **mod_dav** was compiled as a shared object (instead of statically linked directly to the **httpd** binary), you'll need a similar `LoadModule` statement for it, too. Be sure that it comes before the **mod_dav_svn** line:

```
LoadModule dav_module         modules/mod_dav.so
LoadModule dav_svn_module     modules/mod_dav_svn.so
```

At a later location in your configuration file, you now need to tell Apache where you keep your Subversion repository (or repositories). The `Location` directive has an XML-like notation, starting with an opening tag, and ending with a closing tag, with various other configuration directives in the middle. The purpose of the `Location` directive is to instruct Apache to do something special when handling requests that are directed at a given URL or one of its children. In the case of Subversion, you want Apache to simply hand off support for URLs that point at versioned resources to the DAV layer. You can instruct Apache to delegate the handling of all URLs whose path portions (the part of the URL that follows the server's name and the optional port number) begin with `/repos/` to a DAV provider whose repository is located at `/absolute/path/to/repository` using the following `httpd.conf` syntax:

```
<Location /repos>
  DAV svn
  SVNPath /absolute/path/to/repository
</Location>
```

If you plan to support multiple Subversion repositories that will reside in the same parent directory on your local disk, you can use an alternative directive, the `SVNParentPath` directive, to indicate that common parent directory. For example, if you know you will be creating multiple Subversion repositories in a directory `/usr/local/svn` that would be accessed via URLs like `http://my.server.com/svn/repos1`, `http://my.server.com/svn/repos2`, and so on, you could use the `httpd.conf` configuration syntax in the following example:

```
<Location /svn>
  DAV svn

  # any "/svn/foo" URL will map to a repository /usr/local/svn/foo
  SVNParentPath /usr/local/svn
</Location>
```

Using the previous syntax, Apache will delegate the handling of all URLs whose path portions begin with `/svn/` to the Subversion DAV provider, which will then assume that any items in the directory specified by the `SVNParentPath` directive are actually Subversion repositories. This is a particularly convenient syntax in that, unlike the use of the `SVNPath` directive, you don't have to restart Apache in order to create and network new repositories.

Be sure that when you define your new `Location`, it doesn't overlap with other exported Locations. For example, if your main `DocumentRoot` is exported to `/www`, do not export a Subversion repository in `<Location /www/repos>`. If a request comes in for the URI `/www/repos/foo.c`, Apache won't know whether to look for a file `repos/foo.c` in the `DocumentRoot`, or whether to delegate **mod_dav_svn** to return `foo.c` from the Subversion repository. The result is often an error from the server of the form `301 Moved Permanently`.

Server Names and the COPY Request

Subversion makes use of the `COPY` request type to perform server-side copies of files and directories. As part of the sanity checking done by the Apache modules, the source of the copy is expected to be located on the same machine as the destination of the copy. To satisfy this requirement, you might need to tell mod_dav the name you use as the hostname of your server. Generally, you can use the `ServerName` directive in `httpd.conf` to accomplish this.

```
ServerName svn.example.com
```

If you are using Apache's virtual hosting support via the `NameVirtualHost` directive, you may need to use the `ServerAlias` directive to specify additional names that your server is known by. Again, refer to the Apache documentation for full details.

At this stage, you should strongly consider the question of permissions. If you've been running Apache for some time now as your regular web server, you probably already have a collection of content—web pages, scripts and such. These items have already been configured with a set of permissions that allows them to work with Apache, or more appropriately, that allows Apache to work with those files. Apache, when used as a Subversion server, will also need the correct permissions to read and write to your Subversion repository.

You will need to determine a permission system setup that satisfies Subversion's requirements without messing up any previously existing web page or script installations. This might mean changing the permissions on your Subversion repository to match those in use by other things that Apache serves for you, or it could mean using the `User` and `Group` directives in `httpd.conf` to specify that Apache should run as the user and group that owns your Subversion repository. There is no single correct way to set up your permissions, and each administrator will have different reasons for doing things a certain way. Just be aware that permission-related problems are perhaps the most common oversight when configuring a Subversion repository for use with Apache.

Authentication Options

At this point, if you configured `httpd.conf` to contain something like

```
&lt;Location /svn&gt;
```

```
  DAV svn
  SVNParentPath /usr/local/svn
</Location>
```

…then your repository is “anonymously” accessible to the world. Until you configure some authentication and authorization policies, the Subversion repositories you make available via the `Location` directive will be generally accessible to everyone. In other words,

- anyone can use their Subversion client to check out a working copy of a repository URL (or any of its subdirectories),
- anyone can interactively browse the repository's latest revision simply by pointing their web browser to the repository URL, and
- anyone can commit to the repository.

Of course, you might have already set up a `pre-commit` hook script to prevent commits (see the section called “Implementing Repository Hooks”). But as you read on, you'll see that it's also possible use Apache's built-in methods to restrict access in specific ways.

Basic HTTP Authentication

The easiest way to authenticate a client is via the HTTP Basic authentication mechanism, which simply uses a username and password to verify that a user is who she says she is. Apache provides an **htpasswd** utility for managing the list of acceptable usernames and passwords. Let's grant commit access to Sally and Harry. First, we need to add them to the password file.

```
$ ### First time: use -c to create the file
$ ### Use -m to use MD5 encryption of the password, which is more secure
$ htpasswd -cm /etc/svn-auth-file harry
New password: *****
Re-type new password: *****
Adding password for user harry
$ htpasswd -m /etc/svn-auth-file sally
New password: *******
Re-type new password: *******
Adding password for user sally
$
```

Next, you need to add some more `httpd.conf` directives inside your `Location` block to tell Apache what to do with your new password file. The `AuthType` directive specifies the type of authentication system to use. In this case, we want to specify the `Basic` authentication system. `AuthName` is an arbitrary name that you give for the authentication domain. Most browsers will display this name in the pop-up dialog box when the browser is querying the user for his name and password. Finally, use the `AuthUserFile` directive to specify the location of the password file you created using **htpasswd**.

After adding these three directives, your `<Location>` block should look something like this:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
  AuthType Basic
```

```
  AuthName "Subversion repository"
  AuthUserFile /etc/svn-auth-file
</Location>
```

This `<Location>` block is not yet complete, and will not do anything useful. It's merely telling Apache that whenever authorization is required, Apache should harvest a username and password from the Subversion client. What's missing here, however, are directives that tell Apache *which* sorts of client requests require authorization. Wherever authorization is required, Apache will demand authentication as well. The simplest thing to do is protect all requests. Adding `Require valid-user` tells Apache that all requests require an authenticated user:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /etc/svn-auth-file
  Require valid-user
</Location>
```

Be sure to read the next section (the section called “Authorization Options”) for more detail on the `Require` directive and other ways to set authorization policies.

One word of warning: HTTP Basic Auth passwords pass in very nearly plain-text over the network, and thus are extremely insecure. If you're worried about password snooping, it may be best to use some sort of SSL encryption, so that clients authenticate via `https://` instead of `http://`; at a bare minimum, you can configure Apache to use a self-signed server certificate. [5] Consult Apache's documentation (and OpenSSL documentation) about how to do that.

SSL Certificate Management

Businesses that need to expose their repositories for access outside the company firewall should be conscious of the possibility that unauthorized parties could be “sniffing” their network traffic. SSL makes that kind of unwanted attention less likely to result in sensitive data leaks.

If a Subversion client is compiled to use OpenSSL, then it gains the ability to speak to an Apache server via `https://` URLs. The Neon library used by the Subversion client is not only able to verify server certificates, but can also supply client certificates when challenged. When the client and server have exchanged SSL certificates and successfully authenticated one another, all further communication is encrypted via a session key.

It's beyond the scope of this book to describe how to generate client and server certificates, and how to configure Apache to use them. Many other books, including Apache's own documentation, describe this task. But what *can* be covered here is how to manage server and client certificates from an ordinary Subversion client.

When speaking to Apache via `https://`, a Subversion client can receive two different types of information:

- a server certificate
- a demand for a client certificate

[5] While self-signed server certificates are still vulnerable to a “man in the middle” attack, such an attack is much more difficult for a casual observer to pull off, compared to sniffing unprotected passwords.

If the client receives a server certificate, it needs to verify that it trusts the certificate: is the server really who it claims to be? The OpenSSL library does this by examining the signer of the server certificate, or *certifying authority* (CA). If OpenSSL is unable to automatically trust the CA, or if some other problem occurs (such as an expired certificate or hostname mismatch), the Subversion command-line client will ask you whether you want to trust the server certificate anyway:

```
$ svn list https://host.example.com/repos/project

Error validating server certificate for 'https://host.example.com:443':
 - The certificate is not issued by a trusted authority. Use the
   fingerprint to validate the certificate manually!
Certificate information:
 - Hostname: host.example.com
 - Valid: from Jan 30 19:23:56 2004 GMT until Jan 30 19:23:56 2006 GMT
 - Issuer: CA, example.com, Sometown, California, US
 - Fingerprint: 7d:e1:a9:34:33:39:ba:6a:e9:a5:c4:22:98:7b:76:5c:92:a0:9c:7b

(R)eject, accept (t)emporarily or accept (p)ermanently?
```

This dialogue should look familiar; it's essentially the same question you've probably seen coming from your web browser (which is just another HTTP client like Subversion). If you choose the (p)ermanent option, the server certificate will be cached in your private run-time `auth/` area in just the same way your username and password are cached (see the section called “Client Credentials Caching”). If cached, Subversion will automatically trust this certificate in future negotiations.

Your run-time `servers` file also gives you the ability to make your Subversion client automatically trust specific CAs, either globally or on a per-host basis. Simply set the `ssl-authority-files` variable to a semicolon-separated list of PEM-encoded CA certificates:

```
[global]
ssl-authority-files = /path/to/CAcert1.pem;/path/to/CAcert2.pem
```

Many OpenSSL installations also have a pre-defined set of “default” CAs that are nearly universally trusted. To make the Subversion client automatically trust these standard authorities, set the `ssl-trust-default-ca` variable to `true`.

When talking to Apache, a Subversion client might also receive a challenge for a client certificate. Apache is asking the client to identify itself: is the client really who it says it is? If all goes correctly, the Subversion client sends back a private certificate signed by a CA that Apache trusts. A client certificate is usually stored on disk in encrypted format, protected by a local password. When Subversion receives this challenge, it will ask you for both a path to the certificate and the password which protects it:

```
$ svn list https://host.example.com/repos/project

Authentication realm: https://host.example.com:443
Client certificate filename: /path/to/my/cert.p12
Passphrase for '/path/to/my/cert.p12':  ********
…
```

Notice that the client certificate is a “p12” file. To use a client certificate with Subversion, it must be in PKCS#12 format, which is a portable standard. Most web browsers are already

able to import and export certificates in that format. Another option is to use the OpenSSL command-line tools to convert existing certificates into PKCS#12.

Again, the runtime `servers` file allows you to automate this challenge on a per-host basis. Either or both pieces of information can be described in runtime variables:

```
[groups]
examplehost = host.example.com

[examplehost]
ssl-client-cert-file = /path/to/my/cert.p12
ssl-client-cert-password = somepassword
```

Once you've set the `ssl-client-cert-file` and `ssl-client-cert-password` variables, the Subversion client can automatically respond to a client certificate challenge without prompting you. [6]

Authorization Options

At this point, you've configured authentication, but not authorization. Apache is able to challenge clients and confirm identities, but it has not been told how to allow or restrict access to the clients bearing those identities. This section describes two strategies for controlling access to your repositories.

Blanket Access Control

The simplest form of access control is to authorize certain users for either read-only access to a repository, or read/write access to a repository.

You can restrict access on all repository operations by adding the `Require valid-user` directive to your `<Location>` block. Using our previous example, this would mean that only clients that claimed to be either `harry` or `sally`, and provided the correct password for their respective username, would be allowed to do anything with the Subversion repository:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn

  # how to authenticate a user
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file

  # only authenticated users may access the repository
  Require valid-user
</Location>
```

Sometimes you don't need to run such a tight ship. For example, Subversion's own source code repository at http://svn.collab.net/repos/svn allows anyone in the world to perform read-only repository tasks (like checking out working copies and browsing the repository with a web browser), but restricts all write operations to authenticated users. To do this type of selective restriction, you can use the `Limit` and `LimitExcept` configuration directives. Like the `Location` directive, these blocks have starting and ending tags, and you would nest them inside

[6] More security-conscious folk might not want to store the client certificate password in the runtime `servers` file.

your <Location> block.

The parameters present on the Limit and LimitExcept directives are HTTP request types that are affected by that block. For example, if you wanted to disallow all access to your repository except the currently supported read-only operations, you would use the LimitExcept directive, passing the GET, PROPFIND, OPTIONS, and REPORT request type parameters. Then the previously mentioned Require valid-user directive would be placed inside the <LimitExcept> block instead of just inside the <Location> block.

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn

  # how to authenticate a user
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file

  # For any operations other than these, require an authenticated user.
  <LimitExcept GET PROPFIND OPTIONS REPORT>
    Require valid-user
  </LimitExcept>
</Location>
```

These are only a few simple examples. For more in-depth information about Apache access control and the Require directive, take a look at the Security section of the Apache documentation's tutorials collection at http://httpd.apache.org/docs-2.0/misc/tutorials.html.

Per-Directory Access Control

It's possible to set up finer-grained permissions using a second Apache httpd module, **mod_authz_svn**. This module grabs the various opaque URLs passing from client to server, asks **mod_dav_svn** to decode them, and then possibly vetoes requests based on access policies defined in a configuration file.

If you've built Subversion from source code, **mod_authz_svn** is automatically built and installed alongside **mod_dav_svn**. Many binary distributions install it automatically as well. To verify that it's installed correctly, make sure it comes right after **mod_dav_svn**'s LoadModule directive in httpd.conf:

```
LoadModule dav_module         modules/mod_dav.so
LoadModule dav_svn_module     modules/mod_dav_svn.so
LoadModule authz_svn_module   modules/mod_authz_svn.so
```

To activate this module, you need to configure your Location block to use the AuthzSVNAccessFile directive, which specifies a file containing the permissions policy for paths within your repositories. (In a moment, we'll discuss the format of that file.)

Apache is flexible, so you have the option to configure your block in one of three general patterns. To begin, choose one of these basic configuration patterns. (The examples below are very simple; look at Apache's own documentation for much more detail on Apache authentication and authorization options.)

The simplest block is to allow open access to everyone. In this scenario, Apache never sends authentication challenges, so all users are treated as “anonymous”.

Example 6.1. A sample configuration for anonymous access.

```
<Location /repos>
  DAV svn
  SVNParentPath /usr/local/svn

  # our access control policy
  AuthzSVNAccessFile /path/to/access/file
</Location>
```

On the opposite end of the paranoia scale, you can configure your block to demand authentication from everyone. All clients must supply credentials to identify themselves. Your block unconditionally requires authentication via the `Require valid-user` directive, and defines a means to authenticate.

Example 6.2. A sample configuration for authenticated access.

```
<Location /repos>
  DAV svn
  SVNParentPath /usr/local/svn

  # our access control policy
  AuthzSVNAccessFile /path/to/access/file

  # only authenticated users may access the repository
  Require valid-user

  # how to authenticate a user
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file
</Location>
```

A third very popular pattern is to allow a combination of authenticated and anonymous access. For example, many administrators want to allow anonymous users to read certain repository directories, but want only authenticated users to read (or write) more sensitive areas. In this setup, all users start out accessing the repository anonymously. If your access control policy demands a real username at any point, Apache will demand authentication from the client. To do this, you use both the `Satisfy Any` and `Require valid-user` directives together.

Example 6.3. A sample configuration for mixed authenticated/anonymous access.

```
<Location /repos>
  DAV svn
  SVNParentPath /usr/local/svn
```

```
  # our access control policy
  AuthzSVNAccessFile /path/to/access/file

  # try anonymous access first, resort to real
  # authentication if necessary.
  Satisfy Any
  Require valid-user

  # how to authenticate a user
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file
</Location>
```

Once you've settled on one of these three basic `httpd.conf` templates, you need to create your file containing access rules for particular paths within the repository. This is described in the section called “Path-Based Authorization”.

Disabling Path-based Checks

The **mod_dav_svn** module goes through a lot of work to make sure that data you've marked “unreadable” doesn't get accidentally leaked. This means that it needs to closely monitor all of the paths and file-contents returned by commands like **svn checkout** or **svn update** commands. If these commands encounter a path that isn't readable according to some authorization policy, then the path is typically omitted altogether. In the case of history or rename tracing—e.g. running a command like **svn cat -r OLD foo.c** on a file that was renamed long ago—the rename tracking will simply halt if one of the object's former names is determined to be read-restricted.

All of this path-checking can sometimes be quite expensive, especially in the case of **svn log**. When retrieving a list of revisions, the server looks at every changed path in each revision and checks it for readability. If an unreadable path is discovered, then it's omitted from the list of the revision's changed paths (normally seen with the `--verbose` option), and the whole log message is suppressed. Needless to say, this can be time-consuming on revisions that affect a large number of files. This is the cost of security: even if you haven't configured a module like **mod_authz_svn** at all, the **mod_dav_svn** module is still asking Apache **httpd** to run authorization checks on every path. The **mod_dav_svn** module has no idea what authorization modules have been installed, so all it can do is ask Apache to invoke whatever might be present.

On the other hand, there's also an escape-hatch of sorts, one which allows you to trade security features for speed. If you're not enforcing any sort of per-directory authorization (i.e. not using **mod_authz_svn** or similar module), then you can disable all of this path-checking. In your `httpd.conf` file, use the `SVNPathAuthz` directive:

Example 6.4. Disabling path checks altogether

```
<Location /repos>
  DAV svn
  SVNParentPath /usr/local/svn

  SVNPathAuthz off
</Location>
```

The SVNPathAuthz directive is “on” by default. When set “off”, all path-based authorization checking is disabled; **mod_dav_svn** stops invoking authorization checks on every path it discovers.

Extra Goodies

We've covered most of the authentication and authorization options for Apache and mod_dav_svn. But there are a few other nice features that Apache provides.

Repository Browsing

One of the most useful benefits of an Apache/WebDAV configuration for your Subversion repository is that the youngest revisions of your versioned files and directories are immediately available for viewing via a regular web browser. Since Subversion uses URLs to identify versioned resources, those URLs used for HTTP-based repository access can be typed directly into a Web browser. Your browser will issue an HTTP GET request for that URL, and based on whether that URL represents a versioned directory or file, mod_dav_svn will respond with a directory listing or with file contents.

Since the URLs do not contain any information about which version of the resource you wish to see, mod_dav_svn will always answer with the youngest version. This functionality has the wonderful side-effect that you can pass around Subversion URLs to your peers as references to documents, and those URLs will always point at the latest manifestation of that document. Of course, you can even use the URLs as hyperlinks from other web sites, too.

Can I view older revisions?

With an ordinary web browser? In one word: nope. At least, not with **mod_dav_svn** as your only tool.

Your web browser only speaks ordinary HTTP. That means it only knows how to GET public URLs, which represent the latest versions of files and directories. According to the WebDAV/DeltaV specification, each server defines a private URL syntax for older versions of resources, and that syntax is opaque to clients. To find an older version of a file, a client must follow a specific procedure to “discover” the proper URL; the procedure involves issuing a series of WebDAV PROPFIND requests and understanding DeltaV concepts. This is something your web browser simply can't do.

So to answer the question, one obvious way to see older revisions of files and directories is by passing the `--revision (-r)` argument to the **svn list** and **svn cat** commands. To browse old revisions with your web browser, however, you can use third-party software. A good example of this is ViewVC (http://viewvc.tigris.org/). ViewVC was originally written to display CVS repositories through the web, [7] and the latest releases are able to understand Subversion repositories as well.

Proper MIME Type

When browsing a Subversion repository, the web browser gets a clue about how to render a file's contents by looking at the `Content-Type:` header returned in Apache's response to the HTTP GET request. The value of this header is some sort of MIME type. By default, Apache will tell the web browsers that all repository files are of the “default” MIME type, typically `text/plain`. This can be frustrating, however, if a user wishes repository files to render as

[7] Back then, it was called “ViewCVS”.

something more meaningful—for example, it might be nice to have a `foo.html` file in the repository actually render as HTML when browsing.

To make this happen, you only need to make sure that your files have the proper `svn:mime-type` set. This is discussed in more detail in the section called “File Content Type”, and you can even configure your client to automatically attach proper `svn:mime-type` properties to files entering the repository for the first time; see the section called “Automatic Property Setting”.

So in our example, if one were to set the `svn:mime-type` property to `text/html` on file `foo.html`, then Apache would properly tell your web browser to render the file as HTML. One could also attach proper `image/*` mime-type properties to images, and by doing this, ultimately get an entire web site to be viewable directly from a repository! There's generally no problem with doing this, as long as the website doesn't contain any dynamically-generated content.

Customizing the Look

You generally will get more use out of URLs to versioned files—after all, that's where the interesting content tends to lie. But you might have occasion to browse a Subversion directory listing, where you'll quickly note that the generated HTML used to display that listing is very basic, and certainly not intended to be aesthetically pleasing (or even interesting). To enable customization of these directory displays, Subversion provides an XML index feature. A single `SVNIndexXSLT` directive in your repository's `Location` block of `httpd.conf` will instruct mod_dav_svn to generate XML output when displaying a directory listing, and to reference the XSLT stylesheet of your choice:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
  SVNIndexXSLT "/svnindex.xsl"
  …
</Location>
```

Using the `SVNIndexXSLT` directive and a creative XSLT stylesheet, you can make your directory listings match the color schemes and imagery used in other parts of your website. Or, if you'd prefer, you can use the sample stylesheets provided in the Subversion source distribution's `tools/xslt/` directory. Keep in mind that the path provided to the `SVNIndexXSLT` directory is actually a URL path—browsers need to be able to read your stylesheets in order to make use of them!

Listing Repositories

If you're serving a collection of repositories from a single URL via the `SVNParentPath` directive, then it's also possible to have Apache display all available repositories to a web browser. Just activate the `SVNListParentPath` directive:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
  SVNListParentPath on
  …
</Location>
```

If a user now points her web browser to the URL `http://host.example.com/svn/`, she'll see list of all Subversion repositories sitting in `/usr/local/svn`. Obviously, this can be a se-

curity problem, so this feature is turned off by default.

Apache Logging

Because Apache is an HTTP server at heart, it contains fantastically flexible logging features. It's beyond the scope of this book to discuss all ways logging can be configured, but we should point out that even the most generic `httpd.conf` file will cause Apache to produce two logs: `error_log` and `access_log`. These logs may appear in different places, but are typically created in the logging area of your Apache installation. (On Unix, they often live in `/usr/local/apache2/logs/`.)

The `error_log` describes any internal errors that Apache runs into as it works. The `access_log` file records every incoming HTTP request received by Apache. This makes it easy to see, for example, which IP addresses Subversion clients are coming from, how often particular clients use the server, which users are authenticating properly, and which requests succeed or fail.

Unfortunately, because HTTP is a stateless protocol, even the simplest Subversion client operation generates multiple network requests. It's very difficult to look at the `access_log` and deduce what the client was doing—most operations look like a series of cryptic `PROPPATCH`, `GET`, `PUT`, and `REPORT` requests. To make things worse, many client operations send nearly-identical series of requests, so it's even harder to tell them apart.

`mod_dav_svn`, however, can come to your aid. By activating an “operational logging” feature, you can ask `mod_dav_svn` to create a separate log file describing what sort of high-level operations your clients are performing.

To do this, you need to make use of Apache's `CustomLog` directive (which is explained in more detail in Apache's own documentation). Be sure to invoke this directive *outside* of your Subversion `Location` block:

```
<Location /svn>
  DAV svn
  …
</Location>

CustomLog logs/svn_logfile "%t %u %{SVN-ACTION}e" env=SVN-ACTION
```

In this example, we're asking Apache to create a special logfile `svn_logfile` in the standard Apache `logs` directory. The `%t` and `%u` variables are replaced by the time and username of the request, respectively. The really important part are the two instances of `SVN-ACTION`. When Apache sees that variable, it substitutes the value of the `SVN-ACTION` environment variable, which is automatically set by `mod_dav_svn` whenever it detects a high-level client action.

So instead of having to interpret a traditional `access_log` like this:

```
[26/Jan/2007:22:25:29 -0600] "PROPFIND /svn/calc/!svn/vcc/default HTTP/1.1" 207 39
[26/Jan/2007:22:25:29 -0600] "PROPFIND /svn/calc/!svn/bln/59 HTTP/1.1" 207 449
[26/Jan/2007:22:25:29 -0600] "PROPFIND /svn/calc HTTP/1.1" 207 647
[26/Jan/2007:22:25:29 -0600] "REPORT /svn/calc/!svn/vcc/default HTTP/1.1" 200 607
[26/Jan/2007:22:25:31 -0600] "OPTIONS /svn/calc HTTP/1.1" 200 188
[26/Jan/2007:22:25:31 -0600] "MKACTIVITY /svn/calc/!svn/act/e6035ef7-5df0-4ac0-b81
…
```

… you can instead peruse a much more intelligible `svn_logfile` like this:

```
[26/Jan/2007:22:24:20 -0600] - list-dir '/'
[26/Jan/2007:22:24:27 -0600] - update '/'
[26/Jan/2007:22:25:29 -0600] - remote-status '/'
[26/Jan/2007:22:25:31 -0600] sally commit r60
```

Other Features

Several of the features already provided by Apache in its role as a robust Web server can be leveraged for increased functionality or security in Subversion as well. Subversion communicates with Apache using Neon, which is a generic HTTP/WebDAV library with support for such mechanisms as SSL (the Secure Socket Layer, discussed earlier). If your Subversion client is built to support SSL, then it can access your Apache server using `https://`.

Equally useful are other features of the Apache and Subversion relationship, such as the ability to specify a custom port (instead of the default HTTP port 80) or a virtual domain name by which the Subversion repository should be accessed, or the ability to access the repository through an HTTP proxy. These things are all supported by Neon, so Subversion gets that support for free.

Finally, because **mod_dav_svn** is speaking a subset of the WebDAV/DeltaV protocol, it's possible to access the repository via third-party DAV clients. Most modern operating systems (Win32, OS X, and Linux) have the built-in ability to mount a DAV server as a standard network share. This is a complicated topic; for details, read Appendix C, *WebDAV and Autoversioning*.

Path-Based Authorization

Both Apache and **svnserve** are capable of granting (or denying) permissions to users. Typically this is done over the entire repository: a user can read the repository (or not), and she can write to the repository (or not). It's also possible, however, to define finer-grained access rules. One set of users may have permission to write to a certain directory in the repository, but not others; another directory might not even be readable by all but a few special people.

Both servers use a common file format to describe these path-based access rules. In the case of Apache, one needs to load the **mod_authz_svn** module and then add the `AuthzSVNAccessFile` directive (within the `httpd.conf` file) pointing to your own rules-file. (For a full explanation, see the section called “Per-Directory Access Control”.) If you're using **svnserve**, then you need to make the `authz-db` variable (within `svnserve.conf`) point to your rules-file.

Do you really need path-based access control?

A lot of administrators setting up Subversion for the first time tend to jump into path-based access control without giving it a lot of thought. The administrator usually knows which teams of people are working on which projects, so it's easy to jump in and grant certain teams access to certain directories and not others. It seems like a natural thing, and it appeases the administrator's desire to maintain tight control of the repository.

Note, though, that there are often invisible (and visible!) costs associated with this feature. In the visible category, the server needs to do a lot more work to ensure that the user has the right to read or write each specific path; in certain situations, there's very noticeable performance loss. In the invisible category, consider the culture you're creating. Most of the time, while certain users *shouldn't* be committing changes to certain parts of

the repository, that social contract doesn't need to be technologically enforced. Teams can sometimes spontaneously collaborate with each other; someone may want to help someone else out by committing to an area she doesn't normally work on. By preventing this sort of thing at the server level, you're setting up barriers to unexpected collaboration. You're also creating a bunch of rules that need to be maintained as projects develop, new users are added, and so on. It's a bunch of extra work to maintain.

Remember that this is a version control system! Even if somebody accidentally commits a change to something they shouldn't, it's easy to undo the change. And if a user commits to the wrong place with deliberate malice, then it's a social problem anyway, and that the problem needs to be dealt with outside of Subversion.

So before you begin restricting users' access rights, ask yourself if there's a real, honest need for this, or if it's just something that “sounds good” to an administrator. Decide whether it's worth sacrificing some server speed for, and remember that there's very little risk involved; it's bad to become dependent on technology as a crutch for social problems.[8].

As an example to ponder, consider that the Subversion project itself has always had a notion of who is allowed to commit where, but it's always been enforced socially. This is a good model of community trust, especially for open-source projects. Of course, sometimes there *are* truly legitimate needs for path-based access control; within corporations, for example, certain types of data really can be sensitive, and access needs to be genuinely restricted to small groups of people.

Once your server knows where to find your rules-file, it's time to define the rules.

The syntax of the file is the same familiar one used by **svnserve.conf** and the runtime configuration files. Lines that start with a hash (#) are ignored. In its simplest form, each section names a repository and path within it, and the authenticated usernames are the option names within each section. The value of each option describes the user's level of access to the repository path: either `r` (read-only) or `rw` (read-write). If the user is not mentioned at all, no access is allowed.

To be more specific: the value of the section-names are either of the form `[repos-name:path]` or the form `[path]`. If you're using the `SVNParentPath` directive, then it's important to specify the repository names in your sections. If you omit them, then a section like `[/some/dir]` will match the path `/some/dir` in *every* repository. If you're using the `SVNPath` directive, however, then it's fine to only define paths in your sections—after all, there's only one repository.

```
[calc:/branches/calc/bug-142]
harry = rw
sally = r
```

In this first example, the user `harry` has full read and write access on the `/branches/calc/bug-142` directory in the `calc` repository, but the user `sally` has read-only access. Any other users are blocked from accessing this directory.

Of course, permissions are inherited from parent to child directory. That means that we can specify a subdirectory with a different access policy for Sally:

[8] A common theme in this book!

```
[calc:/branches/calc/bug-142]
harry = rw
sally = r

# give sally write access only to the 'testing' subdir
[calc:/branches/calc/bug-142/testing]
sally = rw
```

Now Sally can write to the `testing` subdirectory of the branch, but can still only read other parts. Harry, meanwhile, continues to have complete read-write access to the whole branch.

It's also possible to explicitly deny permission to someone via inheritance rules, by setting the username variable to nothing:

```
[calc:/branches/calc/bug-142]
harry = rw
sally = r

[calc:/branches/calc/bug-142/secret]
harry =
```

In this example, Harry has read-write access to the entire `bug-142` tree, but has absolutely no access at all to the `secret` subdirectory within it.

The thing to remember is that the most specific path always matches first. The server tries to match the path itself, and then the parent of the path, then the parent of that, and so on. The net effect is that mentioning a specific path in the accessfile will always override any permissions inherited from parent directories.

By default, nobody has any access to the repository at all. That means that if you're starting with an empty file, you'll probably want to give at least read permission to all users at the root of the repository. You can do this by using the asterisk variable (`*`), which means “all users”:

```
[/]
* = r
```

This is a common setup; notice that there's no repository name mentioned in the section name. This makes all repositories world readable to all users. Once all users have read-access to the repositories, you can give explicit `rw` permission to certain users on specific subdirectories within specific repositories.

The asterisk variable (`*`) is also worth special mention here: it's the *only* pattern which matches an anonymous user. If you've configured your server block to allow a mixture of anonymous and authenticated access, all users start out accessing anonymously. The server looks for a * value defined for the path being accessed; if it can't find one, then it demands real authentication from the client.

The access file also allows you to define whole groups of users, much like the Unix `/etc/group` file:

```
[groups]
calc-developers = harry, sally, joe
paint-developers = frank, sally, jane
everyone = harry, sally, joe, frank, sally, jane
```

Groups can be granted access control just like users. Distinguish them with an “at” (@) prefix:

```
[calc:/projects/calc]
@calc-developers = rw

[paint:/projects/paint]
@paint-developers = rw
jane = r
```

Groups can also be defined to contain other groups:

```
[groups]
calc-developers = harry, sally, joe
paint-developers = frank, sally, jane
everyone = @calc-developers, @paint-developers
```

Partial Readability and Checkouts

If you're using Apache as your Subversion server and have made certain subdirectories of your repository unreadable to certain users, then you need to be aware of a possible non-optimal behavior with **svn checkout**.

When the client requests a checkout or update over HTTP, it makes a single server request, and receives a single (often large) server response. When the server receives the request, that is the *only* opportunity Apache has to demand user authentication. This has some odd side-effects. For example, if a certain subdirectory of the repository is only readable by user Sally, and user Harry checks out a parent directory, his client will respond to the initial authentication challenge as Harry. As the server generates the large response, there's no way it can re-send an authentication challenge when it reaches the special subdirectory; thus the subdirectory is skipped altogether, rather than asking the user to re-authenticate as Sally at the right moment. In a similar way, if the root of the repository is anonymously world-readable, then the entire checkout will be done without authentication—again, skipping the unreadable directory, rather than asking for authentication partway through.

Supporting Multiple Repository Access Methods

You've seen how a repository can be accessed in many different ways. But is it possible—or safe—for your repository to be accessed by multiple methods simultaneously? The answer is yes, provided you use a bit of foresight.

At any given time, these processes may require read and write access to your repository:

- regular system users using a Subversion client (as themselves) to access the repository directly via `file://` URLs;
- regular system users connecting to SSH-spawned private **svnserve** processes (running as

themselves) which access the repository;

- an **svnserve** process—either a daemon or one launched by **inetd**—running as a particular fixed user;
- an Apache **httpd** process, running as a particular fixed user.

The most common problem administrators run into is repository ownership and permissions. Does every process (or user) in the previous list have the rights to read and write the Berkeley DB files? Assuming you have a Unix-like operating system, a straightforward approach might be to place every potential repository user into a new `svn` group, and make the repository wholly owned by that group. But even that's not enough, because a process may write to the database files using an unfriendly umask—one that prevents access by other users.

So the next step beyond setting up a common group for repository users is to force every repository-accessing process to use a sane umask. For users accessing the repository directly, you can make the **svn** program into a wrapper script that first sets **umask 002** and then runs the real **svn** client program. You can write a similar wrapper script for the **svnserve** program, and add a **umask 002** command to Apache's own startup script, `apachectl`. For example:

```
$ cat /usr/bin/svn

#!/bin/sh

umask 002
/usr/bin/svn-real "$@"
```

Another common problem is often encountered on Unix-like systems. As a repository is used, Berkeley DB occasionally creates new log files to journal its actions. Even if the repository is wholly owned by the **svn** group, these newly created files won't necessarily be owned by that same group, which then creates more permissions problems for your users. A good workaround is to set the group SUID bit on the repository's `db` directory. This causes all newly-created log files to have the same group owner as the parent directory.

Once you've jumped through these hoops, your repository should be accessible by all the necessary processes. It may seem a bit messy and complicated, but the problems of having multiple users sharing write-access to common files are classic ones that are not often elegantly solved.

Fortunately, most repository administrators will never *need* to have such a complex configuration. Users who wish to access repositories that live on the same machine are not limited to using `file://` access URLs—they can typically contact the Apache HTTP server or **svnserve** using `localhost` for the server name in their `http://` or `svn://` URLs. And to maintain multiple server processes for your Subversion repositories is likely to be more of a headache than necessary. We recommend you choose the server that best meets your needs and stick with it!

The svn+ssh:// server checklist

It can be quite tricky to get a bunch of users with existing SSH accounts to share a repository without permissions problems. If you're confused about all the things that you (as an administrator) need to do on a Unix-like system, here's a quick checklist that resummarizes some of things discussed in this section:

- All of your SSH users need to be able to read and write to the repository, so: put all the SSH users into a single group.
- Make the repository wholly owned by that group.
- Set the group permissions to read/write.
- Your users need to use a sane umask when accessing the repository, so: make sure that **svnserve** (`/usr/bin/svnserve`, or wherever it lives in `$PATH`) is actually a wrapper script which sets **umask 002** and executes the real **svnserve** binary.
- Take similar measures when using **svnlook** and **svnadmin**. Either run them with a sane umask, or wrap them as described above.

Chapter 7. Customizing Your Subversion Experience

Version control can be a complex subject, as much art as science, and offering myriad ways of getting stuff done. Throughout this book you've read of the various Subversion command-line client subcommands and the options which modify their behavior. In this chapter, we'll look into still more ways to customize the way Subversion works for you—setting up the Subversion runtime configuration, using external helper applications, Subversion's interaction with the operating system's configured locale, and so on.

Runtime Configuration Area

Subversion provides many optional behaviors that can be controlled by the user. Many of these options are of the kind that a user would wish to apply to all Subversion operations. So, rather than forcing users to remember command-line arguments for specifying these options, and to use them for every operation they perform, Subversion uses configuration files, segregated into a Subversion configuration area.

The Subversion *configuration area* is a two-tiered hierarchy of option names and their values. Usually, this boils down to a special directory that contains *configuration files* (the first tier), which are just text files in standard INI format (with “sections” providing the second tier). These files can be easily edited using your favorite text editor (such as Emacs or vi), and contain directives read by the client to determine which of several optional behaviors the user prefers.

Configuration Area Layout

The first time that the **svn** command-line client is executed, it creates a per-user configuration area. On Unix-like systems, this area appears as a directory named `.subversion` in the user's home directory. On Win32 systems, Subversion creates a folder named `Subversion`, typically inside the `Application Data` area of the user's profile directory (which, by the way, is usually a hidden directory). However, on this platform the exact location differs from system to system, and is dictated by the Windows registry. [1] We will refer to the per-user configuration area using its Unix name, `.subversion`.

In addition to the per-user configuration area, Subversion also recognizes the existence of a system-wide configuration area. This gives system administrators the ability to establish defaults for all users on a given machine. Note that the system-wide configuration area does not alone dictate mandatory policy—the settings in the per-user configuration area override those in the system-wide one, and command-line arguments supplied to the **svn** program have the final word on behavior. On Unix-like platforms, the system-wide configuration area is expected to be the `/etc/subversion` directory; on Windows machines, it looks for a `Subversion` directory inside the common `Application Data` location (again, as specified by the Windows Registry). Unlike the per-user case, the **svn** program does not attempt to create the system-wide configuration area.

The per-user configuration area currently contains three files—two configuration files (`config` and `servers`), and a `README.txt` file which describes the INI format. At the time of their creation, the files contain default values for each of the supported Subversion options, mostly commented out and grouped with textual descriptions about how the values for the key affect Subversion's behavior. To change a certain behavior, you need only to load the appropriate

[1] The `APPDATA` environment variable points to the `Application Data` area, so you can always refer to this folder as `%APPDATA%\Subversion`.

configuration file into a text editor, and modify the desired option's value. If at any time you wish to have the default configuration settings restored, you can simply remove (or rename) your configuration directory and then run some innocuous **svn** command, such as **svn --version**. A new configuration directory with the default contents will be created.

The per-user configuration area also contains a cache of authentication data. The `auth` directory holds a set of subdirectories that contain pieces of cached information used by Subversion's various supported authentication methods. This directory is created in such a way that only the user herself has permission to read its contents.

Configuration and the Windows Registry

In addition to the usual INI-based configuration area, Subversion clients running on Windows platforms may also use the Windows registry to hold the configuration data. The option names and their values are the same as in the INI files. The “file/section” hierarchy is preserved as well, though addressed in a slightly different fashion—in this schema, files and sections are just levels in the registry key tree.

Subversion looks for system-wide configuration values under the `HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion` key. For example, the `global-ignores` option, which is in the `miscellany` section of the `config` file, would be found at `HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Config\Miscellany\global-ignores`. Per-user configuration values should be stored under `HKEY_CURRENT_USER\Software\Tigris.org\Subversion`.

Registry-based configuration options are parsed *before* their file-based counterparts, so are overridden by values found in the configuration files. In other words, Subversion looks for configuration information in the following locations on a Windows system; lower-numbered locations take precedence over higher-numbered locations:

1. Command-line options
2. The per-user INI files
3. The per-user Registry values
4. The system-wide INI files
5. The system-wide Registry values

Also, the Windows Registry doesn't really support the notion of something being “commented out”. However, Subversion will ignore any option key whose name begins with a hash (#) character. This allows you to effectively comment out a Subversion option without deleting the entire key from the Registry, obviously simplifying the process of restoring that option.

The **svn** command-line client never attempts to write to the Windows Registry, and will not attempt to create a default configuration area there. You can create the keys you need using the **REGEDIT** program. Alternatively, you can create a `.reg` file, and then double-click on that file from the Explorer shell, which will cause the data to be merged into your registry.

Example 7.1. Sample Registration Entries (.reg) File.

```
REGEDIT4

[HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Servers\groups]

[HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Servers\global]
"#http-proxy-host"=""
"#http-proxy-port"=""
"#http-proxy-username"=""
"#http-proxy-password"=""
"#http-proxy-exceptions"=""
"#http-timeout"="0"
"#http-compression"="yes"
"#neon-debug-mask"=""
"#ssl-authority-files"=""
"#ssl-trust-default-ca"=""
"#ssl-client-cert-file"=""
"#ssl-client-cert-password"=""

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\auth]
"#store-passwords"="yes"
"#store-auth-creds"="yes"

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\helpers]
"#editor-cmd"="notepad"
"#diff-cmd"=""
"#diff3-cmd"=""
"#diff3-has-program-arg"=""

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\tunnels]

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\miscellany]
"#global-ignores"="*.o *.lo *.la #*# .*.rej *.rej .*~ *~ .#* .DS_Store"
"#log-encoding"=""
"#use-commit-times"=""
"#no-unlock"=""
"#enable-auto-props"=""

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\auto-props]
```

The previous example shows the contents of a `.reg` file which contains some of the most commonly used configuration options and their default values. Note the presence of both system-wide (for network proxy-related options) and per-user settings (editor programs and password storage, among others). Also note that all the options are effectively commented out. You need only to remove the hash (#) character from the beginning of the option names, and set the values as you desire.

Configuration Options

In this section, we will discuss the specific run-time configuration options that are currently supported by Subversion.

Servers

The `servers` file contains Subversion configuration options related to the network layers. There are two special section names in this file—`groups` and `global`. The `groups` section is essentially a cross-reference table. The keys in this section are the names of other sections in the file; their values are *globs*—textual tokens which possibly contain wildcard characters—that are compared against the hostnames of the machine to which Subversion requests are sent.

```
[groups]
beanie-babies = *.red-bean.com
collabnet = svn.collab.net

[beanie-babies]
…

[collabnet]
…
```

When Subversion is used over a network, it attempts to match the name of the server it is trying to reach with a group name under the `groups` section. If a match is made, Subversion then looks for a section in the `servers` file whose name is the matched group's name. From that section it reads the actual network configuration settings.

The `global` section contains the settings that are meant for all of the servers not matched by one of the globs under the `groups` section. The options available in this section are exactly the same as those valid for the other server sections in the file (except, of course, the special `groups` section), and are as follows:

`http-proxy-exceptions`
: This specifies a comma-separated list of patterns for repository hostnames that should accessed directly, without using the proxy machine. The pattern syntax is the same as is used in the Unix shell for filenames. A repository hostname matching any of these patterns will not be proxied.

`http-proxy-host`
: This specifies the hostname of the proxy computer through which your HTTP-based Subversion requests must pass. It defaults to an empty value, which means that Subversion will not attempt to route HTTP requests through a proxy computer, and will instead attempt to contact the destination machine directly.

`http-proxy-port`
: This specifies the port number on the proxy host to use. It defaults to an empty value.

`http-proxy-username`
: This specifies the username to supply to the proxy machine. It defaults to an empty value.

`http-proxy-password`
: This specifies the password to supply to the proxy machine. It defaults to an empty value.

`http-timeout`
: This specifies the amount of time, in seconds, to wait for a server response. If you experience problems with a slow network connection causing Subversion operations to time out, you should increase the value of this option. The default value is `0`, which instructs the underlying HTTP library, Neon, to use its default timeout setting.

`http-compression`
: This specifies whether or not Subversion should attempt to compress network requests made to DAV-ready servers. The default value is `yes` (though compression will only occur if that capability is compiled into the network layer). Set this to `no` to disable compression, such as when debugging network transmissions.

`neon-debug-mask`
: This is an integer mask that the underlying HTTP library, Neon, uses for choosing what type of debugging output to yield. The default value is `0`, which will silence all debugging

output. For more information about how Subversion makes use of Neon, see Chapter 8, *Embedding Subversion*.

`ssl-authority-files`
: This is a semicolon-delimited list of paths to files containing certificates of the certificate authorities (or CAs) that are accepted by the Subversion client when accessing the repository over HTTPS.

`ssl-trust-default-ca`
: Set this variable to `yes` if you want Subversion to automatically trust the set of default CAs that ship with OpenSSL.

`ssl-client-cert-file`
: If a host (or set of hosts) requires an SSL client certificate, you'll normally be prompted for a path to your certificate. By setting this variable to that same path, Subversion will be able to find your client certificate automatically without prompting you. There's no standard place to store your certificate on disk; Subversion will grab it from any path you specify.

`ssl-client-cert-password`
: If your SSL client certificate file is encrypted by a passphrase, Subversion will prompt you for the passphrase whenever the certificate is used. If you find this annoying (and don't mind storing the password in the `servers` file), then you can set this variable to the certificate's passphrase. You won't be prompted anymore.

Config

The `config` file contains the rest of the currently available Subversion run-time options, those not related to networking. There are only a few options in use as of this writing, but they are again grouped into sections in expectation of future additions.

The `auth` section contains settings related to Subversion's authentication and authorization against the repository. It contains:

`store-passwords`
: This instructs Subversion to cache, or not to cache, passwords that are supplied by the user in response to server authentication challenges. The default value is `yes`. Set this to `no` to disable this on-disk password caching. You can override this option for a single instance of the **svn** command using the `--no-auth-cache` command-line parameter (for those subcommands that support it). For more information, see the section called “Client Credentials Caching”.

`store-auth-creds`
: This setting is the same as `store-passwords`, except that it enables or disables disk-caching of *all* authentication information: usernames, passwords, server certificates, and any other types of cacheable credentials.

The `helpers` section controls which external applications Subversion uses to accomplish its tasks. Valid options in this section are:

`editor-cmd`
: This specifies the program Subversion will use to query the user for a log message during a commit operation, such as when using **svn commit** without either the `--message` (`-m`) or `--file` (`-F`) options. This program is also used with the **svn propedit** command—a temporary file is populated with the current value of the property the user wishes to edit,

and the edits take place right in the editor program (see the section called “Properties”). This option's default value is empty. The order of priority for determining the editor command (where lower-numbered locations take precedence over higher-numbered locations) is:

1. Command-line option `--editor-cmd`
2. Environment variable `SVN_EDITOR`
3. Configuration option `editor-cmd`
4. Environment variable `VISUAL`
5. Environment variable `EDITOR`
6. Possibly, a default value built in to Subversion (not present in the official builds)

The value of any of these options or variables is (unlike `diff-cmd`) the beginning of a command line to be executed by the shell. Subversion appends a space and the pathname of the temporary file to be edited. The editor should modify the temporary file and return a zero exit code to indicate success.

`diff-cmd`
: This specifies the absolute path of a differencing program, used when Subversion generates “diff” output (such as when using the **svn diff** command). By default Subversion uses an internal differencing library—setting this option will cause it to perform this task using an external program. See the section called “Using External Differencing Tools” for more details on using such programs.

`diff3-cmd`
: This specifies the absolute path of a three-way differencing program. Subversion uses this program to merge changes made by the user with those received from the repository. By default Subversion uses an internal differencing library—setting this option will cause it to perform this task using an external program. See the section called “Using External Differencing Tools” for more details on using such programs.

`diff3-has-program-arg`
: This flag should be set to `true` if the program specified by the `diff3-cmd` option accepts a `--diff-program` command-line parameter.

The `tunnels` section allows you to define new tunnel schemes for use with **svnserve** and `svn://` client connections. For more details, see the section called “Tunneling over SSH”.

The `miscellany` section is where everything that doesn't belong elsewhere winds up. [2] In this section, you can find:

`global-ignores`
: When running the **svn status** command, Subversion lists unversioned files and directories along with the versioned ones, annotating them with a ? character (see the section called “See an overview of your changes”). Sometimes, it can be annoying to see uninteresting, unversioned items—for example, object files that result from a program's compilation—in this display. The `global-ignores` option is a list of whitespace-delimited globs which describe the names of files and directories that Subversion should not display unless they are versioned. The default value is `*.o *.lo *.la #*# .*.rej *.rej .*~ *~ .#* .DS_Store`.

[2] Anyone for potluck dinner?

As well as **svn status**, the **svn add** and **svn import** commands also ignore files that match the list when they are scanning a directory. You can override this behaviour for a single instance of any of these commands by explicitly specifying the file name, or by using the `--no-ignore` command-line flag.

For information on more fine-grained control of ignored items, see the section called “Ignoring Unversioned Items”.

`enable-auto-props`
: This instructs Subversion to automatically set properties on newly added or imported files. The default value is `no`, so set this to `yes` to enable Auto-props. The `auto-props` section of this file specifies which properties are to be set on which files.

`log-encoding`
: This variable sets the default character set encoding for commit log messages. It's a permanent form of the `--encoding` option (see the section called “**svn** Options”). The Subversion repository stores log messages in UTF-8, and assumes that your log message is written using your operating system's native locale. You should specify a different encoding if your commit messages are written in any other encoding.

`use-commit-times`
: Normally your working copy files have timestamps that reflect the last time they were touched by any process, whether that be your own editor or by some **svn** subcommand. This is generally convenient for people developing software, because build systems often look at timestamps as a way of deciding which files need to be recompiled.

  In other situations, however, it's sometimes nice for the working copy files to have timestamps that reflect the last time they were changed in the repository. The **svn export** command always places these “last-commit timestamps” on trees that it produces. By setting this config variable to `yes`, the **svn checkout**, **svn update**, **svn switch**, and **svn revert** commands will also set last-commit timestamps on files that they touch.

The `auto-props` section controls the Subversion client's ability to automatically set properties on files when they are added or imported. It contains any number of key-value pairs in the format `PATTERN = PROPNAME=PROPVALUE` where `PATTERN` is a file pattern that matches a set of filenames and the rest of the line is the property and its value. Multiple matches on a file will result in multiple propsets for that file; however, there is no guarantee that auto-props will be applied in the order in which they are listed in the config file, so you can't have one rule “override” another. You can find several examples of auto-props usage in the `config` file. Lastly, don't forget to set `enable-auto-props` to `yes` in the `miscellany` section if you want to enable auto-props.

Localization

Localization is the act of making programs behave in a region-specific way. When a program formats numbers or dates in a way specific to your part of the world, or prints messages (or accepts input) in your native language, the program is said to be *localized*. This section describes steps Subversion has made towards localization.

Understanding locales

Most modern operating systems have a notion of the “current locale”—that is, the region or country whose localization conventions are honored. These conventions—typically chosen by some runtime configuration mechanism on the computer—affect the way in which programs present data to the user, as well as the way in which they accept user input.

On most Unix-like systems, you can check the values of the locale-related runtime configuration options by running the **locale** command:

```
$ locale
LANG=
LC_COLLATE="C"
LC_CTYPE="C"
LC_MESSAGES="C"
LC_MONETARY="C"
LC_NUMERIC="C"
LC_TIME="C"
LC_ALL="C"
```

The output is a list of locale-related environment variables and their current values. In this example, the variables are all set to the default `C` locale, but users can set these variables to specific country/language code combinations. For example, if one were to set the `LC_TIME` variable to `fr_CA`, then programs would know to present time and date information formatted according a French-speaking Canadian's expectations. And if one were to set the `LC_MESSAGES` variable to `zh_TW`, then programs would know to present human-readable messages in Traditional Chinese. Setting the `LC_ALL` variable has the effect of changing every locale variable to the same value. The value of `LANG` is used as a default value for any locale variable that is unset. To see the list of available locales on a Unix system, run the command **locale -a**.

On Windows, locale configuration is done via the “Regional and Language Options” control panel item. There you can view and select the values of individual settings from the available locales, and even customize (at a sickening level of detail) several of the display formatting conventions.

Subversion's use of locales

The Subversion client, **svn**, honors the current locale configuration in two ways. First, it notices the value of the `LC_MESSAGES` variable and attempts to print all messages in the specified language. For example:

```
$ export LC_MESSAGES=de_DE
$ svn help cat
cat: Gibt den Inhalt der angegebenen Dateien oder URLs aus.
Aufruf: cat ZIEL[@REV]...
…
```

This behavior works identically on both Unix and Windows systems. Note, though, that while your operating system might have support for a certain locale, the Subversion client still may not be able to speak the particular language. In order to produce localized messages, human volunteers must provide translations for each language. The translations are written using the GNU gettext package, which results in translation modules that end with the `.mo` filename extension. For example, the German translation file is named `de.mo`. These translation files are installed somewhere on your system. On Unix, they typically live in `/usr/share/locale/`, while on Windows they're often found in the `\share\locale\` folder in Subversion's installation area. Once installed, a module is named after the program it provides translations for. For example, the `de.mo` file may ultimately end up installed as `/usr/share/locale/de/LC_MESSAGES/subversion.mo`. By browsing the installed `.mo` files, you can see which languages the Subversion client is able to speak.

The second way in which the locale is honored involves how **svn** interprets your input. The repository stores all paths, filenames, and log messages in Unicode, encoded as UTF-8. In that

sense, the repository is *internationalized*—that is, the repository is ready to accept input in any human language. This means, however, that the Subversion client is responsible for sending only UTF-8 filenames and log messages into the repository. In order to do this, it must convert the data from the native locale into UTF-8.

For example, suppose you create a file named `caffè.txt`, and then when committing the file, you write the log message as “Adesso il caffè è più forte”. Both the filename and log message contain non-ASCII characters, but because your locale is set to `it_IT`, the Subversion client knows to interpret them as Italian. It uses an Italian character set to convert the data to UTF-8 before sending them off to the repository.

Note that while the repository demands UTF-8 filenames and log messages, it *does not* pay attention to file contents. Subversion treats file contents as opaque strings of bytes, and neither client nor server makes an attempt to understand the character set or encoding of the contents.

Character set conversion errors

While using Subversion, you might get hit with an error related to character set conversions:

```
svn: Can't convert string from native encoding to 'UTF-8':
…
svn: Can't convert string from 'UTF-8' to native encoding:
…
```

Errors like this typically occur when the Subversion client has received a UTF-8 string from the repository, but not all of the characters in that string can be represented using the encoding of the current locale. For example, if your locale is `en_US` but a collaborator has committed a Japanese filename, you're likely to see this error when you receive the file during an **svn update**.

The solution is either to set your locale to something which *can* represent the incoming UTF-8 data, or to change the filename or log message in the repository. (And don't forget to slap your collaborator's hand—projects should decide on common languages ahead of time, so that all participants are using the same locale.)

Using External Differencing Tools

The presence of `--diff-cmd` and `--diff3-cmd` options, and similarly named runtime configuration parameters (see the section called “Config”), can lead to a false notion of how easy it is to use external differencing (or “diff”) and merge tools with Subversion. While Subversion can use most of popular such tools available, the effort invested in setting this up often turns out to be non-trivial.

The interface between Subversion and external diff and merge tools harkens back to a time when Subversion's only contextual differencing capabilities were built around invocations of the GNU diffutils toolchain, specifically the **diff** and **diff3** utilities. To get the kind of behavior Subversion needed, it called these utilities with more than a handful of options and parameters, most of which were quite specific to the utilities. Some time later, Subversion grew its own internal differencing library, and as a failover mechanism, [3] the `--diff-cmd` and `--diff3-cmd` options were added to the Subversion command-line client so users could more easily indicate

[3] Subversion developers are good, but even the best make mistakes.

that they preferred to use the GNU diff and diff3 utilities instead of the newfangled internal diff library. If those options were used, Subversion would simply ignore the internal diff library, and fall back to running those external programs, lengthy argument lists and all. And that's where things remain today.

It didn't take long for folks to realize that having such easy configuration mechanisms for specifying that Subversion should use the external GNU diff and diff3 utilities located at a particular place on the system could be applied toward the use of other diff and merge tools, too. After all, Subversion didn't actually verify that the things it was being told to run were members of the GNU diffutils toolchain. But the only configurable aspect of using those external tools is their location on the system—not the option set, parameter order, etc. Subversion continues throwing all those GNU utility options at your external diff tool regardless of whether or not that program can understand those options. And that's where things get unintuitive for most users.

The key to using external diff and merge tools (other than GNU diff and diff3, of course) with Subversion is to use wrapper scripts which convert the input from Subversion into something that your differencing tool can understand, and then to convert the output of your tool back into a format which Subversion expects—the format that the GNU tools would have used. The following sections cover the specifics of those expectations.



The decision on when to fire off a contextual diff or merge as part of a larger Subversion operation is made entirely by Subversion, and is affected by, among other things, whether or not the files being operated on are human-readable as determined by their `svn:mime-type` property. This means, for example, that even if you had the niftiest Microsoft Word-aware differencing or merging tool in the Universe, it would never be invoked by Subversion so long as your versioned Word documents had a configured MIME type that denoted that they were not human-readable (such as `application/msword`). For more about MIME type settings, see the section called “File Content Type”

External diff

Subversion calls external diff programs with parameters suitable for the GNU diff utility, and expects only that the external program return with a successful error code. For most alternative diff programs, only the sixth and seventh arguments—the paths of the files which represent the left and right sides of the diff, respectively—are of interest. Note that Subversion runs the diff program once per modified file covered by the Subversion operation, so if your program runs in an asynchronous fashion (or “backgrounded”), you might have several instances of it all running simultaneously. Finally, Subversion expects that your program return an error code of 1 if your program detected differences, or 0 if it did not—any other error code is considered a fatal error. [4]

Example 7.2, “diffwrap.sh” and Example 7.3, “diffwrap.bat” are templates for external diff tool wrappers in the Bourne shell and Windows batch scripting languages, respectively.

Example 7.2. diffwrap.sh

```
#!/bin/sh

# Configure your favorite diff program here.
```

[4] The GNU diff manual page puts it this way: “An exit status of 0 means no differences were found, 1 means some differences were found, and 2 means trouble.”

```
DIFF="/usr/local/bin/my-diff-tool"

# Subversion provides the paths we need as the sixth and seventh
# parameters.
LEFT=${6}
RIGHT=${7}

# Call the diff command (change the following line to make sense for
# your merge program).
$DIFF --left $LEFT --right $RIGHT

# Return an errorcode of 0 if no differences were detected, 1 if some were.
# Any other errorcode will be treated as fatal.
```

Example 7.3. diffwrap.bat

```
@ECHO OFF

REM Configure your favorite diff program here.
SET DIFF="C:\Program Files\Funky Stuff\My Diff Tool.exe"

REM Subversion provides the paths we need as the sixth and seventh
REM parameters.
SET LEFT=%6
SET RIGHT=%7

REM Call the diff command (change the following line to make sense for
REM your merge program).
%DIFF% --left %LEFT% --right %RIGHT%

REM Return an errorcode of 0 if no differences were detected, 1 if some were.
REM Any other errorcode will be treated as fatal.
```

External diff3

Subversion calls external merge programs with parameters suitable for the GNU diff3 utility, expecting that the external program return with a successful error code and that the full file contents which result from the completed merge operation are printed on the standard output stream (so that Subversion can redirect them into the appropriate version controlled file). For most alternative merge programs, only the ninth, tenth, and eleventh arguments, the paths of the files which represent the “mine”, “older”, and “yours” inputs, respectively, are of interest. Note that because Subversion depends on the output of your merge program, you wrapper script must not exit before that output has been delivered to Subversion. When it finally does exit, it should return an error code of 0 if the merge was successful, or 1 if unresolved conflicts remain in the output—any other error code is considered a fatal error.

Example 7.4, “diff3wrap.sh” and Example 7.5, “diff3wrap.bat” are templates for external merge tool wrappers in the Bourne shell and Windows batch scripting languages, respectively.

Example 7.4. diff3wrap.sh

```
#!/bin/sh

# Configure your favorite diff3/merge program here.
DIFF3="/usr/local/bin/my-merge-tool"

# Subversion provides the paths we need as the ninth, tenth, and eleventh
# parameters.
MINE=${9}
OLDER=${10}
YOURS=${11}

# Call the merge command (change the following line to make sense for
# your merge program).
$DIFF3 --older $OLDER --mine $MINE --yours $YOURS

# After performing the merge, this script needs to print the contents
# of the merged file to stdout.  Do that in whatever way you see fit.
# Return an errorcode of 0 on successful merge, 1 if unresolved conflicts
# remain in the result.  Any other errorcode will be treated as fatal.
```

Example 7.5. diff3wrap.bat

```
@ECHO OFF

REM Configure your favorite diff3/merge program here.
SET DIFF3="C:\Program Files\Funky Stuff\My Merge Tool.exe"

REM Subversion provides the paths we need as the ninth, tenth, and eleventh
REM parameters.  But we only have access to nine parameters at a time, so we
REM shift our nine-parameter window twice to let us get to what we need.
SHIFT
SHIFT
SET MINE=%7
SET OLDER=%8
SET YOURS=%9

REM Call the merge command (change the following line to make sense for
REM your merge program).
%DIFF3% --older %OLDER% --mine %MINE% --yours %YOURS%

REM After performing the merge, this script needs to print the contents
REM of the merged file to stdout.  Do that in whatever way you see fit.
REM Return an errorcode of 0 on successful merge, 1 if unresolved conflicts
REM remain in the result.  Any other errorcode will be treated as fatal.
```

Chapter 8. Embedding Subversion

Subversion has a modular design: it's implemented as a collection of libraries written in C. Each library has a well-defined purpose and Application Programming Interface (API), and that interface is available not only for Subversion itself to use, but for any software that wishes to embed or otherwise programmatically control Subversion. Additionally, Subversion's API is available not only to other C programs, but also to programs written in higher-level languages such as Python, Perl, Java, or Ruby.

This chapter is for those who wish to interact with Subversion through its public API or its various language bindings. If you wish to write robust wrapper scripts around Subversion functionality to simplify your own life, are trying to develop more complex integrations between Subversion and other pieces of software, or just have an interest in Subversion's various library modules and what they offer, this chapter is for you. If, however, you don't foresee yourself participating with Subversion at such a level, feel free to skip this chapter with the confidence that your experience as a Subversion user will not be affected.

Layered Library Design

Each of Subversion's core libraries can be said to exist in one of three main layers—the Repository Layer, the Repository Access (RA) Layer, or the Client Layer (see Figure 1, “Subversion's Architecture”). We will examine these layers shortly, but first, let's briefly summarize Subversion's various libraries. For the sake of consistency, we will refer to the libraries by their extensionless Unix library names (libsvn_fs, libsvn_wc, mod_dav_svn, etc.).

libsvn_client
: Primary interface for client programs

libsvn_delta
: Tree and byte-stream differencing routines

libsvn_diff
: Contextual differencing and merging routines

libsvn_fs
: Filesystem commons and module loader

libsvn_fs_base
: The Berkeley DB filesystem back-end

libsvn_fs_fs
: The native filesystem (FSFS) back-end

libsvn_ra
: Repository Access commons and module loader

libsvn_ra_dav
: The WebDAV Repository Access module

libsvn_ra_local
: The local Repository Access module

libsvn_ra_serf
: Another (experimental) WebDAV Repository Access module

libsvn_ra_svn
: The custom protocol Repository Access module

libsvn_repos
: Repository interface

libsvn_subr
: Miscellaneous helpful subroutines

libsvn_wc
: The working copy management library

mod_authz_svn
: Apache authorization module for Subversion repositories access via WebDAV

mod_dav_svn
: Apache module for mapping WebDAV operations to Subversion ones

The fact that the word “miscellaneous” only appears once in the previous list is a good sign. The Subversion development team is serious about making sure that functionality lives in the right layer and libraries. Perhaps the greatest advantage of the modular design is its lack of complexity from a developer's point of view. As a developer, you can quickly formulate that kind of “big picture” that allows you to pinpoint the location of certain pieces of functionality with relative ease.

Another benefit of modularity is the ability to replace a given module with a whole new library that implements the same API without affecting the rest of the code base. In some sense, this happens within Subversion already. The libsvn_ra_dav, libsvn_ra_local, libsvn_ra_serf, and libsvn_ra_svn libraries each implement the same interface, all working as plugins to libsvn_ra. And all four communicate with the Repository Layer—libsvn_ra_local connects to the repository directly; the other three do so over a network. The libsvn_fs_base and libsvn_fs_fs libraries are another pair of libraries that implement the same functionality in different ways—both are plugins to the common libsvn_fs library.

The client itself also highlights the benefits of modularity in the Subversion design. Subversion's libsvn_client library is a one-stop shop for most of the functionality necessary for designing a working Subversion client (see the section called “Client Layer”). So while the Subversion distribution provides only the **svn** command-line client program, there are several third-party programs which provide various forms of graphical client UI. These GUIs use the same APIs that the stock command-line client does. This type of modularity has played a large role in the proliferation of available Subversion clients and IDE integrations and, by extension, to the tremendous adoption rate of Subversion itself.

Repository Layer

When referring to Subversion's Repository Layer, we're generally talking about two basic concepts—the versioned filesystem implementation (accessed via libsvn_fs, and supported by its libsvn_fs_base and libsvn_fs_fs plugins), and the repository logic that wraps it (as implemented in libsvn_repos). These libraries provide the storage and reporting mechanisms for the various revisions of your version-controlled data. This layer is connected to the Client Layer via the Repository Access Layer, and is, from the perspective of the Subversion user, the stuff at the “other end of the line.”

The Subversion Filesystem is not a kernel-level filesystem that one would install in an operating system (like the Linux ext2 or NTFS), but a virtual filesystem. Rather than storing “files” and “directories” as real files and directories (as in, the kind you can navigate through using your

favorite shell program), it uses one of two available abstract storage backends—either a Berkeley DB database environment, or a flat-file representation. (To learn more about the two repository back-ends, see the section called “Choosing a Data Store”.) There has even been considerable interest by the development community in giving future releases of Subversion the ability to use other back-end database systems, perhaps through a mechanism such as Open Database Connectivity (ODBC). In fact, Google did something similar to this before launching the Google Code Project Hosting service: they announced in mid-2006 that members of its Open Source team had written a new proprietary Subversion filesystem plugin which used their ultra-scalable Bigtable database for its storage.

The filesystem API exported by libsvn_fs contains the kinds of functionality you would expect from any other filesystem API—you can create and remove files and directories, copy and move them around, modify file contents, and so on. It also has features that are not quite as common, such as the ability to add, modify, and remove metadata (“properties”) on each file or directory. Furthermore, the Subversion Filesystem is a versioning filesystem, which means that as you make changes to your directory tree, Subversion remembers what your tree looked like before those changes. And before the previous changes. And the previous ones. And so on, all the way back through versioning time to (and just beyond) the moment you first started adding things to the filesystem.

All the modifications you make to your tree are done within the context of a Subversion commit transaction. The following is a simplified general routine for modifying your filesystem:

1. Begin a Subversion commit transaction.

2. Make your changes (adds, deletes, property modifications, etc.).

3. Commit your transaction.

Once you have committed your transaction, your filesystem modifications are permanently stored as historical artifacts. Each of these cycles generates a single new revision of your tree, and each revision is forever accessible as an immutable snapshot of “the way things were.”

The Transaction Distraction

The notion of a Subversion transaction can become easily confused with the transaction support provided by the underlying database itself, especially given the former's close proximity to the Berkeley DB database code in libsvn_fs_base. Both types of transaction exist to provide atomicity and isolation. In other words, transactions give you the ability to perform a set of actions in an all-or-nothing fashion—either all the actions in the set complete with success, or they all get treated as if *none* of them ever happened—and in a way that does not interfere with other processes acting on the data.

Database transactions generally encompass small operations related specifically to the modification of data in the database itself (such as changing the contents of a table row). Subversion transactions are larger in scope, encompassing higher-level operations like making modifications to a set of files and directories which are intended to be stored as the next revision of the filesystem tree. If that isn't confusing enough, consider the fact that Subversion uses a database transaction during the creation of a Subversion transaction (so that if the creation of Subversion transaction fails, the database will look as if we had never attempted that creation in the first place)!

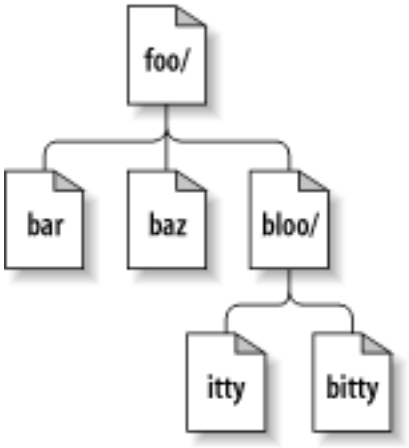
Fortunately for users of the filesystem API, the transaction support provided by the database system itself is hidden almost entirely from view (as should be expected from a

properly modularized library scheme). It is only when you start digging into the implementation of the filesystem itself that such things become visible (or interesting).

Most of the functionality provided by the filesystem interface deals with actions that occur on individual filesystem paths. That is, from outside of the filesystem, the primary mechanism for describing and accessing the individual revisions of files and directories comes through the use of path strings like `/foo/bar`, just as if you were addressing files and directories through your favorite shell program. You add new files and directories by passing their paths-to-be to the right API functions. You query for information about them by the same mechanism.

Unlike most filesystems, though, a path alone is not enough information to identify a file or directory in Subversion. Think of a directory tree as a two-dimensional system, where a node's siblings represent a sort of left-and-right motion, and descending into subdirectories a downward motion. Figure 8.1, “Files and directories in two dimensions” shows a typical representation of a tree as exactly that.

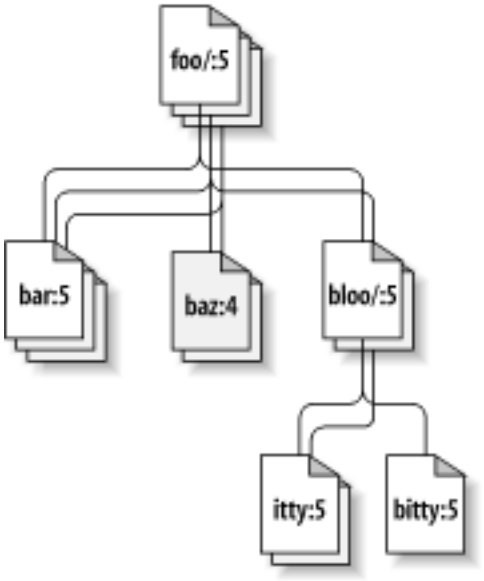
Figure 8.1. Files and directories in two dimensions



The difference here is that the Subversion filesystem has a nifty third dimension that most filesystems do not have—Time! [1] In the filesystem interface, nearly every function that has a *`path`* argument also expects a *`root`* argument. This svn_fs_root_t argument describes either a revision or a Subversion transaction (which is simply a revision-in-the-making), and provides that third-dimensional context needed to understand the difference between `/foo/bar` in revision 32, and the same path as it exists in revision 98. Figure 8.2, “Versioning time—the third dimension!” shows revision history as an added dimension to the Subversion filesystem universe.

Figure 8.2. Versioning time—the third dimension!

[1] We understand that this may come as a shock to sci-fi fans who have long been under the impression that Time was actually the *fourth* dimension, and we apologize for any emotional trauma induced by our assertion of a different theory.



As we mentioned earlier, the libsvn_fs API looks and feels like any other filesystem, except that it has this wonderful versioning capability. It was designed to be usable by any program interested in a versioning filesystem. Not coincidentally, Subversion itself is interested in that functionality. But while the filesystem API should be sufficient for basic file and directory versioning support, Subversion wants more—and that is where libsvn_repos comes in.

The Subversion repository library (libsvn_repos) sits (logically speaking) atop the libsvn_fs API, providing additional functionality beyond that of the underlying versioned filesystem logic. It does not completely wrap each and every filesystem function—only certain major steps in the general cycle of filesystem activity are wrapped by the repository interface. Some of these include the creation and commit of Subversion transactions, and the modification of revision properties. These particular events are wrapped by the repository layer because they have hooks associated with them. A repository hook system is not strictly related to implementing a versioning filesystem, so it lives in the repository wrapper library.

The hooks mechanism is but one of the reasons for the abstraction of a separate repository library from the rest of the filesystem code. The libsvn_repos API provides several other important utilities to Subversion. These include the abilities to:

- create, open, destroy, and perform recovery steps on a Subversion repository and the filesystem included in that repository.
- describe the differences between two filesystem trees.
- query for the commit log messages associated with all (or some) of the revisions in which a set of files was modified in the filesystem.
- generate a human-readable “dump” of the filesystem, a complete representation of the revisions in the filesystem.
- parse that dump format, loading the dumped revisions into a different Subversion repository.

As Subversion continues to evolve, the repository library will grow with the filesystem library to offer increased functionality and configurable option support.

Repository Access Layer

If the Subversion Repository Layer is at “the other end of the line”, the Repository Access (RA) Layer is the line itself. Charged with marshaling data between the client libraries and the repository, this layer includes the libsvn_ra module loader library, the RA modules themselves (which currently includes libsvn_ra_dav, libsvn_ra_local, libsvn_ra_serf, and libsvn_ra_svn), and any additional libraries needed by one or more of those RA modules (such as the mod_dav_svn Apache module or libsvn_ra_svn's server, **svnserve**).

Since Subversion uses URLs to identify its repository resources, the protocol portion of the URL scheme (usually `file://`, `http://`, `https://`, `svn://`, or `svn+ssh://`) is used to determine which RA module will handle the communications. Each module registers a list of the protocols it knows how to “speak” so that the RA loader can, at runtime, determine which module to use for the task at hand. You can determine which RA modules are available to the Subversion command-line client, and what protocols they claim to support, by running **svn --version**:

```
$ svn --version
svn, version 1.4.3 (r23084)
   compiled Jan 18 2007, 07:47:40

Copyright (C) 2000-2006 CollabNet.
Subversion is open source software, see http://subversion.tigris.org/
This product includes software developed by CollabNet (http://www.Collab.Net/).

The following repository access (RA) modules are available:

* ra_dav : Module for accessing a repository via WebDAV (DeltaV) protocol.
  - handles 'http' scheme
  - handles 'https' scheme
* ra_svn : Module for accessing a repository using the svn network protocol.
  - handles 'svn' scheme
* ra_local : Module for accessing a repository on local disk.
  - handles 'file' scheme

$
```

The public API exported by the RA Layer contains functionality necessary for sending and receiving versioned data to and from the repository. And each of the available RA plugins is able to perform that task using a specific protocol—libsvn_ra_dav speaks HTTP/WebDAV (optionally using SSL encryption) with an Apache HTTP Server that is running the mod_dav_svn Subversion server module; libsvn_ra_svn speaks a custom network protocol with the **svnserve** program; and so on.

And for those who wish to access a Subversion repository using still another protocol, that is precisely why the Repository Access Layer is modularized! Developers can simply write a new library that implements the RA interface on one side and communicates with the repository on the other. Your new library can use existing network protocols, or you can invent your own. You could use inter-process communication (IPC) calls, or—let's get crazy, shall we?—you could even implement an email-based protocol. Subversion supplies the APIs; you supply the creativity.

Client Layer

On the client side, the Subversion working copy is where all the action takes place. The bulk of functionality implemented by the client-side libraries exists for the sole purpose of managing working copies—directories full of files and other subdirectories which serve as a sort of local,

editable “reflection” of one or more repository locations—and propagating changes to and from the Repository Access layer.

Subversion's working copy library, libsvn_wc, is directly responsible for managing the data in the working copies. To accomplish this, the library stores administrative information about each working copy directory within a special subdirectory. This subdirectory, named `.svn`, is present in each working copy directory and contains various other files and directories which record state and provide a private workspace for administrative action. For those familiar with CVS, this `.svn` subdirectory is similar in purpose to the CVS administrative directories found in CVS working copies. For more information about the `.svn` administrative area, see the section called “Inside the Working Copy Administration Area”in this chapter.

The Subversion client library, libsvn_client, has the broadest responsibility; its job is to mingle the functionality of the working copy library with that of the Repository Access Layer, and then to provide the highest-level API to any application that wishes to perform general revision control actions. For example, the function `svn_client_checkout()` takes a URL as an argument. It passes this URL to the RA layer and opens an authenticated session with a particular repository. It then asks the repository for a certain tree, and sends this tree into the working copy library, which then writes a full working copy to disk (`.svn` directories and all).

The client library is designed to be used by any application. While the Subversion source code includes a standard command-line client, it should be very easy to write any number of GUI clients on top of the client library. New GUIs (or any new client, really) for Subversion need not be clunky wrappers around the included command-line client—they have full access via the libsvn_client API to same functionality, data, and callback mechanisms that the command-line client uses. In fact, the Subversion source code tree contains a small C program (which can be found at `tools/examples/minimal_client.c` that exemplifies how to wield the Subversion API to create a simple client program

Binding Directly—A Word About Correctness

Why should your GUI program bind directly with a libsvn_client instead of acting as a wrapper around a command-line program? Besides simply being more efficient, it can be more correct as well. A command-line program (like the one supplied with Subversion) that binds to the client library needs to effectively translate feedback and requested data bits from C types to some form of human-readable output. This type of translation can be lossy. That is, the program may not display all of the information harvested from the API, or may combine bits of information for compact representation.

If you wrap such a command-line program with yet another program, the second program has access only to already-interpreted (and as we mentioned, likely incomplete) information, which it must *again* translate into *its* representation format. With each layer of wrapping, the integrity of the original data is potentially tainted more and more, much like the result of making a copy of a copy (of a copy …) of a favorite audio or video cassette.

But the most compelling argument for binding directly to the APIs instead of wrapping other programs is that the Subversion project makes compatibility promises regarding its APIs. Across minor versions of those APIs (such as between 1.3 and 1.4), no function's prototype will change. In other words, you aren't forced to update your program's source code simply because you've upgraded to a new version of Subversion. Certain functions might be deprecated, but they still work, and this gives you a buffer of time to eventually embrace the newer APIs. These kinds of compatibility promises do not exist for Subversion command-line program output, which is subject to change from release to release.

Inside the Working Copy Administration Area

As we mentioned earlier, each directory of a Subversion working copy contains a special subdirectory called `.svn` which houses administrative data about that working copy directory. Subversion uses the information in `.svn` to keep track of things like:

- Which repository location(s) are represented by the files and subdirectories in the working copy directory.
- What revision of each of those files and directories are currently present in the working copy.
- Any user-defined properties that might be attached to those files and directories.
- Pristine (un-edited) copies of the working copy files.

The Subversion working copy administration area's layout and contents are considered implementation details not really intended for human consumption. Developers are encouraged to use Subversion's public APIs, or the tools that Subversion provides, to access and manipulate the working copy data, instead of directly reading or modifying those files. The file formats employed by the working copy library for its administrative data do change from time to time—a fact that the public APIs do a great job of hiding from the average user. In this section, we expose some of these implementation details sheerly to appease your overwhelming curiosity.

The Entries File

Perhaps the single most important file in the `.svn` directory is the `entries` file. It contains the bulk of the administrative information about the versioned items in a working copy directory. It is this one file which tracks the repository URLs, pristine revision, file checksums, pristine text and property timestamps, scheduling and conflict state information, last-known commit information (author, revision, timestamp), local copy history—practically everything that a Subversion client is interested in knowing about a versioned (or to-be-versioned) resource!

Folks familiar with CVS's administrative directories will have recognized at this point that Subversion's `.svn/entries` file serves the purposes of, among other things, CVS's `CVS/Entries`, `CVS/Root`, and `CVS/Repository` files combined.

The format of the `.svn/entries` file has changed over time. Originally an XML file, it now uses a custom—though still human-readable—file format. While XML was a great choice for early developers of Subversion who were frequently debugging the file's contents (and Subversion's behavior in light of them), the need for easy developer debugging has diminished as Subversion has matured, and has been replaced by the user's need for snappier performance. Be aware that Subversion's working copy library automatically upgrades working copies from one format to another—it reads the old formats, and writes the new—which saves you the hassle of checking out a new working copy, but can also complicate situations where different versions of Subversion might be trying to use the same working copy.

Pristine Copies and Property Files

As mentioned before, the `.svn` directory also holds the pristine “text-base” versions of files. Those can be found in `.svn/text-base`. The benefits of these pristine copies are multiple—network-free checks for local modifications and difference reporting, network-free reversion of modified or missing files, more efficient transmission of changes to the server—but comes at the cost of having each versioned file stored at least twice on disk. These days, this seems to be a negligible penalty for most files. However, the situation gets uglier as the size of

your versioned files grows. Some attention is being given to making the presence of the “text-base” an option. Ironically though, it is as your versioned files' sizes get larger that the existence of the “text-base” becomes more crucial—who wants to transmit a huge file across a network just because they want to commit a tiny change to it?

Similar in purpose to the “text-base” files are the property files and their pristine “prop-base” copies, located in `.svn/props` and `.svn/prop-base` respectively. Since directories can have properties, too, there are also `.svn/dir-props` and `.svn/dir-prop-base` files.

Using the APIs

Developing applications against the Subversion library APIs is fairly straightforward. Subversion is primarily a set of C libraries, with header (.h) files that live in the `subversion/include` directory of the source tree. These headers are copied into your system locations (for example, `/usr/local/include`) when you build and install Subversion itself from source. These headers represent the entirety of the functions and types meant to be accessible by users of the Subversion libraries. The Subversion developer community is meticulous about ensuring that the public API is well-documented—refer directly to the header files for that documentation.

When examining the public header files, the first thing you might notice is that Subversion's datatypes and functions are namespace protected. That is, every public Subversion symbol name begins with `svn_`, followed by a short code for the library in which the symbol is defined (such as `wc`, `client`, `fs`, etc.), followed by a single underscore (_) and then the rest of the symbol name. Semi-public functions (used among source files of a given library but not by code outside that library, and found inside the library directories themselves) differ from this naming scheme in that instead of a single underscore after the library code, they use a double underscore (__). Functions that are private to a given source file have no special prefixing, and are declared `static`. Of course, a compiler isn't interested in these naming conventions, but they help to clarify the scope of a given function or datatype.

Another good source of information about programming against the Subversion APIs is the project's own hacking guidelines, which can be found at http://subversion.tigris.org/hacking.html. This document contains useful information which, while aimed at developers and would-be developers of Subversion itself, is equally applicable to folks developing against Subversion as a set of third-party libraries. [2]

The Apache Portable Runtime Library

Along with Subversion's own datatypes, you will see many references to datatypes that begin with `apr_`—symbols from the Apache Portable Runtime (APR) library. APR is Apache's portability library, originally carved out of its server code as an attempt to separate the OS-specific bits from the OS-independent portions of the code. The result was a library that provides a generic API for performing operations that differ mildly—or wildly—from OS to OS. While the Apache HTTP Server was obviously the first user of the APR library, the Subversion developers immediately recognized the value of using APR as well. This means that there is practically no OS-specific code in Subversion itself. Also, it means that the Subversion client compiles and runs anywhere that Apache HTTP Server itself does. Currently this list includes all flavors of Unix, Win32, BeOS, OS/2, and Mac OS X.

In addition to providing consistent implementations of system calls that differ across operating systems, [3] APR gives Subversion immediate access to many custom datatypes, such as dynamic arrays and hash tables. Subversion uses these types extensively. But perhaps the most pervasive APR datatype, found in nearly every Subversion API prototype, is the

[2] After all, Subversion uses Subversion's APIs, too.
[3] Subversion uses ANSI system calls and datatypes as much as possible.

apr_pool_t—the APR memory pool. Subversion uses pools internally for all its memory allocation needs (unless an external library requires a different memory management mechanism for data passed through its API), [4] and while a person coding against the Subversion APIs is not required to do the same, they *are* required to provide pools to the API functions that need them. This means that users of the Subversion API must also link against APR, must call `apr_initialize()` to initialize the APR subsystem, and then must create and manage pools for use with Subversion API calls, typically by using `svn_pool_create()`, `svn_pool_clear()`, and `svn_pool_destroy()`.

Programming with Memory Pools

Almost every developer who has used the C programming language has at some point sighed at the daunting task of managing memory usage. Allocating enough memory to use, keeping track of those allocations, freeing the memory when you no longer need it—these tasks can be quite complex. And of course, failure to do those things properly can result in a program that crashes itself, or worse, crashes the computer.

Higher-level languages, on the other hand, take the job of memory management away from the developer completely. [5] Languages like Java and Python use *garbage collection*, allocating memory for objects when needed, and automatically freeing that memory when the object is no longer in use.

APR provides a middle-ground approach called pool-based memory management. It allows the developer to control memory usage at a lower resolution—per chunk (or “pool”) of memory, instead of per allocated object. Rather than using `malloc()` and friends to allocate enough memory for a given object, you ask APR to allocate the memory from a memory pool. When you're finished using the objects you've created in the pool, you destroy the entire pool, effectively de-allocating the memory consumed by *all* the objects you allocated from it. Thus, rather than keeping track of individual objects which need to be de-allocated, your program simply considers the general lifetimes of those objects, and allocates the objects in a pool whose lifetime (the time between the pool's creation and its deletion) matches the object's needs.

URL and Path Requirements

With remote version control operation as the whole point of Subversion's existence, it makes sense that some attention has been paid to internationalization (i18n) support. After all, while “remote” might mean “across the office”, it could just as well mean “across the globe.” To facilitate this, all of Subversion's public interfaces that accept path arguments expect those paths to be canonicalized, and encoded in UTF-8. This means, for example, that any new client binary that drives the libsvn_client interface needs to first convert paths from the locale-specific encoding to UTF-8 before passing those paths to the Subversion libraries, and then re-convert any resultant output paths from Subversion back into the locale's encoding before using those paths for non-Subversion purposes. Fortunately, Subversion provides a suite of functions (see `subversion/include/svn_utf.h`) that can be used by any program to do these conversions.

Also, Subversion APIs require all URL parameters to be properly URI-encoded. So, instead of passing `file:///home/username/My File.txt` as the URL of a file named `My File.txt`, you need to pass `file:///home/username/My%20File.txt`. Again, Subversion supplies helper functions that your application can use—`svn_path_uri_encode()` and `svn_path_uri_decode()`, for URI encoding and decoding, respectively.

[4] Neon and Berkeley DB are examples of such libraries.
[5] Or at least make it something you only toy with when doing extremely tight program optimization.

Using Languages Other than C and C++

If you are interested in using the Subversion libraries in conjunction with something other than a C program—say a Python or Perl script—Subversion has some support for this via the Simplified Wrapper and Interface Generator (SWIG). The SWIG bindings for Subversion are located in `subversion/bindings/swig`. They are still maturing, but they are usable. These bindings allow you to call Subversion API functions indirectly, using wrappers that translate the datatypes native to your scripting language into the datatypes needed by Subversion's C libraries.

Significant efforts have been made towards creating functional SWIG-generated bindings for Python, Perl, and Ruby. To some extent, the work done preparing the SWIG interface files for these languages is reusable in efforts to generate bindings for other languages supported by SWIG (which include versions of C#, Guile, Java, MzScheme, OCaml, PHP, and Tcl, among others). However, some extra programming is required to compensate for complex APIs that SWIG needs some help translating between languages. For more information on SWIG itself, see the project's website at http://www.swig.org/.

Subversion also has language bindings for Java. The JavaJL bindings (located in `subversion/bindings/java` in the Subversion source tree) aren't SWIG-based, but are instead a mixture of javah and hand-coded JNI. JavaHL most covers Subversion client-side APIs, and is specifically targeted at implementors of Java-based Subversion clients and IDE integrations.

Subversion's language bindings tend to lack the level of developer attention given to the core Subversion modules, but can generally be trusted as production-ready. A number of scripts and applications, alternative Subversion GUI clients and other third-party tools are successfully using Subversion's language bindings today to accomplish their Subversion integrations.

It's worth noting here that there are other options for interfacing with Subversion using other languages: alternative bindings for Subversion which aren't provided by the Subversion development community at all. You can find links to these alternative bindings on the Subversion project's links page (at http://subversion.tigris.org/links.html), but there are a couple of popular ones we feel are especially noteworthy. First, Barry Scott's PySVN bindings (http://pysvn.tigris.org/) are a popular option for binding with Python. PySVN boasts of a more Pythonic interface than the more C-like APIs provided by Subversion's own Python bindings. For folks looking for a pure Java implementation of Subversion, check out SVNKit (http://svnkit.com/), which is Subversion re-written from the ground up in Java. You should exercise caution here, though—because SVNKit doesn't use the core Subversion libraries, it's behavior is not guaranteed to match that of Subversion itself.

Code Samples

Example 8.1, “Using the Repository Layer” contains a code segment (written in C) that illustrates some of the concepts we've been discussing. It uses both the repository and filesystem interfaces (as can be determined by the prefixes `svn_repos_` and `svn_fs_` of the function names, respectively) to create a new revision in which a directory is added. You can see the use of an APR pool, which is passed around for memory allocation purposes. Also, the code reveals a somewhat obscure fact about Subversion error handling—all Subversion errors must be explicitly handled to avoid memory leakage (and in some cases, application failure).

Example 8.1. Using the Repository Layer

```
/* Convert a Subversion error into a simple boolean error code.
 *
```

```
 * NOTE:  Subversion errors must be cleared (using svn_error_clear())
 *        because they are allocated from the global pool, else memory
 *        leaking occurs.
 */
#define INT_ERR(expr)                                   \
  do {                                                  \
    svn_error_t *__temperr = (expr);                    \
    if (__temperr)                                      \
      {                                                 \
        svn_error_clear(__temperr);                     \
        return 1;                                       \
      }                                                 \
    return 0;                                           \
  } while (0)

/* Create a new directory at the path NEW_DIRECTORY in the Subversion
 * repository located at REPOS_PATH.  Perform all memory allocation in
 * POOL.  This function will create a new revision for the addition of
 * NEW_DIRECTORY.  Return zero if the operation completes
 * successfully, non-zero otherwise.
 */
static int
make_new_directory(const char *repos_path,
                   const char *new_directory,
                   apr_pool_t *pool)
{
  svn_error_t *err;
  svn_repos_t *repos;
  svn_fs_t *fs;
  svn_revnum_t youngest_rev;
  svn_fs_txn_t *txn;
  svn_fs_root_t *txn_root;
  const char *conflict_str;

  /* Open the repository located at REPOS_PATH.
   */
  INT_ERR(svn_repos_open(&repos, repos_path, pool));

  /* Get a pointer to the filesystem object that is stored in REPOS.
   */
  fs = svn_repos_fs(repos);

  /* Ask the filesystem to tell us the youngest revision that
   * currently exists.
   */
  INT_ERR(svn_fs_youngest_rev(&youngest_rev, fs, pool));

  /* Begin a new transaction that is based on YOUNGEST_REV.  We are
   * less likely to have our later commit rejected as conflicting if we
   * always try to make our changes against a copy of the latest snapshot
   * of the filesystem tree.
   */
  INT_ERR(svn_fs_begin_txn(&txn, fs, youngest_rev, pool));

  /* Now that we have started a new Subversion transaction, get a root
   * object that represents that transaction.
   */
  INT_ERR(svn_fs_txn_root(&txn_root, txn, pool));

  /* Create our new directory under the transaction root, at the path
   * NEW_DIRECTORY.
   */
  INT_ERR(svn_fs_make_dir(txn_root, new_directory, pool));
```

```
  /* Commit the transaction, creating a new revision of the filesystem
   * which includes our added directory path.
   */
  err = svn_repos_fs_commit_txn(&conflict_str, repos,
                                &youngest_rev, txn, pool);
  if (! err)
    {
      /* No error?  Excellent!  Print a brief report of our success.
       */
      printf("Directory '%s' was successfully added as new revision "
             "'%ld'.\n", new_directory, youngest_rev);
    }
  else if (err->apr_err == SVN_ERR_FS_CONFLICT)
    {
      /* Uh-oh.  Our commit failed as the result of a conflict
       * (someone else seems to have made changes to the same area
       * of the filesystem that we tried to modify).  Print an error
       * message.
       */
      printf("A conflict occurred at path '%s' while attempting "
             "to add directory '%s' to the repository at '%s'.\n",
             conflict_str, new_directory, repos_path);
    }
  else
    {
      /* Some other error has occurred.  Print an error message.
       */
      printf("An error occurred while attempting to add directory '%s' "
             "to the repository at '%s'.\n",
             new_directory, repos_path);
    }

  INT_ERR(err);
}
```

Note that in Example 8.1, “Using the Repository Layer”, the code could just as easily have committed the transaction using `svn_fs_commit_txn()`. But the filesystem API knows nothing about the repository library's hook mechanism. If you want your Subversion repository to automatically perform some set of non-Subversion tasks every time you commit a transaction (like, for example, sending an email that describes all the changes made in that transaction to your developer mailing list), you need to use the libsvn_repos-wrapped version of that function, which adds the hook triggering functionality—in this case, `svn_repos_fs_commit_txn()`. (For more information regarding Subversion's repository hooks, see the section called “Implementing Repository Hooks”.)

Now let's switch languages. Example 8.2, “Using the Repository Layer with Python” is a sample program that uses Subversion's SWIG Python bindings to recursively crawl the youngest repository revision, and print the various paths reached during the crawl.

Example 8.2. Using the Repository Layer with Python

```
#!/usr/bin/python

"""Crawl a repository, printing versioned object path names."""

import sys
import os.path
```

```
import svn.fs, svn.core, svn.repos

def crawl_filesystem_dir(root, directory):
    """Recursively crawl DIRECTORY under ROOT in the filesystem, and return
    a list of all the paths at or below DIRECTORY."""

    # Print the name of this path.
    print directory + "/"

    # Get the directory entries for DIRECTORY.
    entries = svn.fs.svn_fs_dir_entries(root, directory)

    # Loop over the entries.
    names = entries.keys()
    for name in names:
        # Calculate the entry's full path.
        full_path = directory + '/' + name

        # If the entry is a directory, recurse.  The recursion will return
        # a list with the entry and all its children, which we will add to
        # our running list of paths.
        if svn.fs.svn_fs_is_dir(root, full_path):
            crawl_filesystem_dir(root, full_path)
        else:
            # Else it's a file, so print its path here.
            print full_path

def crawl_youngest(repos_path):
    """Open the repository at REPOS_PATH, and recursively crawl its
    youngest revision."""

    # Open the repository at REPOS_PATH, and get a reference to its
    # versioning filesystem.
    repos_obj = svn.repos.svn_repos_open(repos_path)
    fs_obj = svn.repos.svn_repos_fs(repos_obj)

    # Query the current youngest revision.
    youngest_rev = svn.fs.svn_fs_youngest_rev(fs_obj)

    # Open a root object representing the youngest (HEAD) revision.
    root_obj = svn.fs.svn_fs_revision_root(fs_obj, youngest_rev)

    # Do the recursive crawl.
    crawl_filesystem_dir(root_obj, "")

if __name__ == "__main__":
    # Check for sane usage.
    if len(sys.argv) != 2:
        sys.stderr.write("Usage: %s REPOS_PATH\n"
                         % (os.path.basename(sys.argv[0])))
        sys.exit(1)

    # Canonicalize the repository path.
    repos_path = svn.core.svn_path_canonicalize(sys.argv[1])

    # Do the real work.
    crawl_youngest(repos_path)
```

This same program in C would need to deal with APR's memory pool system. But Python handles memory usage automatically, and Subversion's Python bindings adhere to that convention. In C, you'd be working with custom datatypes (such as those provided by the APR library) for representing the hash of entries and the list of paths, but Python has hashes (called

“dictionaries”) and lists as built-in datatypes, and provides a rich collection of functions for operating on those types. So SWIG (with the help of some customizations in Subversion's language bindings layer) takes care of mapping those custom datatypes into the native datatypes of the target language. This provides a more intuitive interface for users of that language.

The Subversion Python bindings can be used for working copy operations, too. In the previous section of this chapter, we mentioned the `libsvn_client` interface, and how it exists for the sole purpose of simplifying the process of writing a Subversion client. Example 8.3, “A Python Status Crawler” is a brief example of how that library can be accessed via the SWIG Python bindings to recreate a scaled-down version of the **svn status** command.

Example 8.3. A Python Status Crawler

```
#!/usr/bin/env python

"""Crawl a working copy directory, printing status information."""

import sys
import os.path
import getopt
import svn.core, svn.client, svn.wc

def generate_status_code(status):
    """Translate a status value into a single-character status code,
    using the same logic as the Subversion command-line client."""
    code_map = { svn.wc.svn_wc_status_none        : ' ',
                 svn.wc.svn_wc_status_normal      : ' ',
                 svn.wc.svn_wc_status_added       : 'A',
                 svn.wc.svn_wc_status_missing     : '!',
                 svn.wc.svn_wc_status_incomplete  : '!',
                 svn.wc.svn_wc_status_deleted     : 'D',
                 svn.wc.svn_wc_status_replaced    : 'R',
                 svn.wc.svn_wc_status_modified    : 'M',
                 svn.wc.svn_wc_status_merged      : 'G',
                 svn.wc.svn_wc_status_conflicted  : 'C',
                 svn.wc.svn_wc_status_obstructed  : '~',
                 svn.wc.svn_wc_status_ignored     : 'I',
                 svn.wc.svn_wc_status_external    : 'X',
                 svn.wc.svn_wc_status_unversioned : '?',
               }
    return code_map.get(status, '?')

def do_status(wc_path, verbose):
    # Calculate the length of the input working copy path.
    wc_path_len = len(wc_path)

    # Build a client context baton.
    ctx = svn.client.svn_client_ctx_t()

    def _status_callback(path, status, root_path_len=wc_path_len):
        """A callback function for svn_client_status."""

        # Print the path, minus the bit that overlaps with the root of
        # the status crawl
        text_status = generate_status_code(status.text_status)
        prop_status = generate_status_code(status.prop_status)
        print '%s%s  %s' % (text_status, prop_status, path[wc_path_len + 1:])

    # Do the status crawl, using _status_callback() as our callback function.
```

```
    svn.client.svn_client_status(wc_path, None, _status_callback,
                                 1, verbose, 0, 0, ctx)

def usage_and_exit(errorcode):
    """Print usage message, and exit with ERRORCODE."""
    stream = errorcode and sys.stderr or sys.stdout
    stream.write("""Usage: %s OPTIONS WC-PATH
Options:
  --help, -h    : Show this usage message
  --verbose, -v : Show all statuses, even uninteresting ones
""" % (os.path.basename(sys.argv[0])))
    sys.exit(errorcode)

if __name__ == '__main__':
    # Parse command-line options.
    try:
        opts, args = getopt.getopt(sys.argv[1:], "hv", ["help", "verbose"])
    except getopt.GetoptError:
        usage_and_exit(1)
    verbose = 0
    for opt, arg in opts:
        if opt in ("-h", "--help"):
            usage_and_exit(0)
        if opt in ("-v", "--verbose"):
            verbose = 1
    if len(args) != 1:
        usage_and_exit(2)

    # Canonicalize the repository path.
    wc_path = svn.core.svn_path_canonicalize(args[0])

    # Do the real work.
    try:
        do_status(wc_path, verbose)
    except svn.core.SubversionException, e:
        sys.stderr.write("Error (%d): %s\n" % (e[1], e[0]))
        sys.exit(1)
```

As was the case in Example 8.2, “Using the Repository Layer with Python”, this program is pool-free and uses, for the most part, normal Python data types. The call to `svn_client_ctx_t()` is deceiving because the public Subversion API has no such function—this just happens to be a case where SWIG's automatic language generation bleeds through a little bit (the function is a sort of factory function for Python's version of the corresponding complex C structure). Also note that the path passed to this program (like the last one) gets run through `svn_path_canonicalize()`, because to *not* do so runs the risk of triggering the underlying Subversion C library's assertions about such things, which translate into rather immediate and unceremonious program abortion.

Chapter 9. Subversion Complete Reference

This chapter is intended to be a complete reference to using Subversion. This includes the command line client (**svn**) and all its subcommands, as well as the repository administration programs (**svnadmin** and **svnlook**) and their respective subcommands.

The Subversion Command Line Client: svn

To use the command line client, you type **svn**, the subcommand you wish to use [1], and any options or targets that you wish to operate on—there is no specific order that the subcommand and the options must appear in. For example, all of the following are valid ways to use **svn status**:

```
$ svn -v status
$ svn status -v
$ svn status -v myfile
```

You can find many more examples of how to use most client commands in Chapter 2, *Basic Usage* and commands for managing properties in the section called “Properties”.

svn Options

While Subversion has different options for its subcommands, all options are global—that is, each option is guaranteed to mean the same thing regardless of the subcommand you use it with. For example, `--verbose` (`-v`) always means “verbose output”, regardless of the subcommand you use it with.

`--auto-props`
: Enables auto-props, overriding the `enable-auto-props` directive in the `config` file.

`--change` (`-c`) *`ARG`*
: Used as a means to refer to a specific “change” (aka a revision), this option is syntactic sugar for “-r ARG-1:ARG”.

`--config-dir` *`DIR`*
: Instructs Subversion to read configuration information from the specified directory instead of the default location (`.subversion` in the user's home directory).

`--diff-cmd` *`CMD`*
: Specifies an external program to use to show differences between files. When **svn diff** is invoked without this option, it uses Subversion's internal diff engine, which provides unified diffs by default. If you want to use an external diff program, use `--diff-cmd`. You can pass options to the diff program with the `--extensions` option (more on that later in this section).

`--diff3-cmd` *`CMD`*
: Specifies an external program to use to merge files.

[1] Yes, yes, you don't need a subcommand to use the `--version` option, but we'll get to that in just a minute.

`--dry-run`
Goes through all the motions of running a command, but makes no actual changes—either on disk or in the repository.

`--editor-cmd` *`CMD`*
Specifies an external program to use to edit a log message or a property value. See the `editor-cmd` section in the section called “Config” for ways to specify a default editor.

`--encoding` *`ENC`*
Tells Subversion that your commit message is encoded in the charset provided. The default is your operating system's native locale, and you should specify the encoding if your commit message is in any other encoding.

`--extensions` (`-x`) *`ARGS`*
Specifies an argument or arguments that Subversion should pass to an external diff command. This option is valid only when used with the **svn diff** or **svn merge** commands, with the `--diff-cmd` option. If you wish to pass multiple arguments, you must enclose all of them in quotes (for example, **svn diff --diff-cmd /usr/bin/diff -x "-b -E"**).

`--file` (`-F`) *`FILENAME`*
Uses the contents of the named file for the specified subcommand, though different subcommands do different things with this content. For example, **svn commit** uses the content as a commit log, whereas **svn propset** uses it as a property value.

`--force`
Forces a particular command or operation to run. There are some operations that Subversion will prevent you from doing in normal usage, but you can pass the force option to tell Subversion “I know what I'm doing as well as the possible repercussions of doing it, so let me at 'em”. This option is the programmatic equivalent of doing your own electrical work with the power on—if you don't know what you're doing, you're likely to get a nasty shock.

`--force-log`
Forces a suspicious parameter passed to the `--message` (`-m`) or `--file` (`-F`) options to be accepted as valid. By default, Subversion will produce an error if parameters to these options look like they might instead be targets of the subcommand. For example, if you pass a versioned file's path to the `--file` (`-F`) option, Subversion will assume you've made a mistake, that the path was instead intended as the target of the operation, and that you simply failed to provide some other—unversioned—file as the source of your log message. To assert your intent and override these types of errors, pass the `--force-log` option to subcommands that accept log messages.

`--help` (`-h` or `-?`)
If used with one or more subcommands, shows the built-in help text for each subcommand. If used alone, it displays the general client help text.

`--ignore-ancestry`
Tells Subversion to ignore ancestry when calculating differences (rely on path contents alone).

`--ignore-externals`
Tells Subversion to ignore external definitions and the external working copies managed by them.

`--incremental`
Prints output in a format suitable for concatenation.

`--limit` *`NUM`*

Show only the first *NUM* log messages.

`--message` (`-m`) *MESSAGE*
Indicates that you will specify a either a log message or a lock comment on the command line, following this option. For example:

```
$ svn commit -m "They don't make Sunday."
```

`--new` *ARG*
Uses *ARG* as the newer target (for use with **svn diff**).

`--no-auth-cache`
Prevents caching of authentication information (e.g. username and password) in the Subversion administrative directories.

`--no-auto-props`
Disables auto-props, overriding the `enable-auto-props` directive in the `config` file.

`--no-diff-added`
Prevents Subversion from printing differences for added files. The default behavior when you add a file is for **svn diff** to print the same differences that you would see if you had added the entire contents of an existing (empty) file.

`--no-diff-deleted`
Prevents Subversion from printing differences for deleted files. The default behavior when you remove a file is for **svn diff** to print the same differences that you would see if you had left the file but removed all the content.

`--no-ignore`
Shows files in the status listing that would normally be omitted since they match a pattern in the `global-ignores` configuration option or the `svn:ignore` property. See the section called “Config” and the section called “Ignoring Unversioned Items” for more information.

`--no-unlock`
Don't automatically unlock files (the default commit behavior is to unlock all files listed as part of the commit). See the section called “Locking” for more information.

`--non-interactive`
In the case of an authentication failure, or insufficient credentials, prevents prompting for credentials (e.g. username or password). This is useful if you're running Subversion inside of an automated script and it's more appropriate to have Subversion fail than to prompt for more information.

`--non-recursive` (`-N`)
Stops a subcommand from recursing into subdirectories. Most subcommands recurse by default, but some subcommands—usually those that have the potential to remove or undo your local modifications—do not.

`--notice-ancestry`
Pay attention to ancestry when calculating differences.

`--old` *ARG*
Uses *ARG* as the older target (for use with **svn diff**).

`--password` *PASS*
Indicates that you are providing your password for authentication on the command

line—otherwise, if it is needed, Subversion will prompt you for it.

`--quiet` (`-q`)
Requests that the client print only essential information while performing an operation.

`--recursive` (`-R`)
Makes a subcommand recurse into subdirectories. Most subcommands recurse by default.

`--relocate` *`FROM TO [PATH...]`*
Used with the **svn switch** subcommand, changes the location of the repository that your working copy references. This is useful if the location of your repository changes and you have an existing working copy that you'd like to continue to use. See **svn switch** for an example.

`--revision` (`-r`) *`REV`*
Indicates that you're going to supply a revision (or range of revisions) for a particular operation. You can provide revision numbers, revision keywords or dates (in curly braces), as arguments to the revision option. If you wish to provide a range of revisions, you can provide two revisions separated by a colon. For example:

```
$ svn log -r 1729
$ svn log -r 1729:HEAD
$ svn log -r 1729:1744
$ svn log -r {2001-12-04}:{2002-02-17}
$ svn log -r 1729:{2002-02-17}
```

See the section called “Revision Keywords” for more information.

`--revprop`
Operates on a revision property instead of a property specific to a file or directory. This option requires that you also pass a revision with the `--revision` (`-r`) option.

`--show-updates` (`-u`)
Causes the client to display information about which files in your working copy are out-of-date. This doesn't actually update any of your files—it just shows you which files will be updated if you run **svn update**.

`--stop-on-copy`
Causes a Subversion subcommand which is traversing the history of a versioned resource to stop harvesting that historical information when a copy—that is, a location in history where that resource was copied from another location in the repository—is encountered.

`--strict`
Causes Subversion to use strict semantics, a notion which is rather vague unless talking about specific subcommands (namely, **svn propget**).

`--targets` *`FILENAME`*
Tells Subversion to get the list of files that you wish to operate on from the filename you provide instead of listing all the files on the command line.

`--username` *`NAME`*
Indicates that you are providing your username for authentication on the command line—otherwise, if it is needed, Subversion will prompt you for it.

`--verbose` (`-v`)
Requests that the client print out as much information as it can while running any subcommand. This may result in Subversion printing out additional fields, detailed information

about every file, or additional information regarding its actions.

`--version`
: Prints the client version info. This information not only includes the version number of the client, but also a listing of all repository access modules that the client can use to access a Subversion repository. With `--quiet` (`-q`) it prints only the version number in a compact form.

`--xml`
: Prints output in XML format.

svn Subcommands

Here are the various subcommands:

Name

svn add — Add files, directories, or symbolic links.

Synopsis

```
svn add PATH...
```

Description

Schedule files, directories, or symbolic links in your working copy for addition to the repository. They will be uploaded and added to the repository on your next commit. If you add something and change your mind before committing, you can unschedule the addition using **svn revert**.

Alternate Names

None

Changes

Working Copy

Accesses Repository

No

Options

```
--targets FILENAME
--non-recursive (-N)
--quiet (-q)
--config-dir DIR
--no-ignore
--auto-props
--no-auto-props
--force
```

Examples

To add a file to your working copy:

```
$ svn add foo.c
A         foo.c
```

When adding a directory, the default behavior of **svn add** is to recurse:

```
$ svn add testdir
A         testdir
A         testdir/a
A         testdir/b
A         testdir/c
A         testdir/d
```

You can add a directory without adding its contents:

```
$ svn add --non-recursive otherdir
A         otherdir
```

Normally, the command **svn add *** will skip over any directories that are already under version control. Sometimes, however, you may want to add every unversioned object in your working copy, including those hiding deeper down. Passing the `--force` option makes **svn add** recurse into versioned directories:

```
$ svn add * --force
A         foo.c
A         somedir/bar.c
A         otherdir/docs/baz.doc
…
```

Name

svn blame — Show author and revision information in-line for the specified files or URLs.

Synopsis

```
svn blame TARGET[@REV]...
```

Description

Show author and revision information in-line for the specified files or URLs. Each line of text is annotated at the beginning with the author (username) and the revision number for the last change to that line.

Alternate Names

praise, annotate, ann

Changes

Nothing

Accesses Repository

Yes

Options

```
--revision (-r) ARG
--verbose (-v)
--incremental
--xml
--extensions (-x) ARG
--force
--username ARG
--password ARG
--no-auth-cache
--non-interactive
--config-dir ARG
```

Examples

If you want to see blame annotated source for `readme.txt` in your test repository:

```
$ svn blame http://svn.red-bean.com/repos/test/readme.txt
     3      sally This is a README file.
     5      harry You should read this.
```

Even if **svn blame** says that Harry last modified readme.txt in revision 5, you'll have to examine exactly what the revision changed to be sure that Harry changed the *context* of the line—he may have just adjusted the whitespace.

Name

svn cat — Output the contents of the specified files or URLs.

Synopsis

```
svn cat TARGET[@REV]...
```

Description

Output the contents of the specified files or URLs. For listing the contents of directories, see **svn list**.

Alternate Names

None

Changes

Nothing

Accesses Repository

Yes

Options

```
--revision (-r) REV
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

If you want to view readme.txt in your repository without checking it out:

```
$ svn cat http://svn.red-bean.com/repos/test/readme.txt
This is a README file.
You should read this.
```



If your working copy is out of date (or you have local modifications) and you want to see the HEAD revision of a file in your working copy, **svn cat** will automatically fetch the HEAD revision when you give it a path:

```
$ cat foo.c
This file is in my local working copy
and has changes that I've made.

$ svn cat foo.c
```

```
Latest revision fresh from the repository!
```

Name

svn checkout — Check out a working copy from a repository.

Synopsis

```
svn checkout URL[@REV]... [PATH]
```

Description

Check out a working copy from a repository. If *PATH* is omitted, the basename of the URL will be used as the destination. If multiple URLs are given each will be checked out into a subdirectory of *PATH*, with the name of the subdirectory being the basename of the URL.

Alternate Names

co

Changes

Creates a working copy.

Accesses Repository

Yes

Options

```
--revision (-r) REV
--quiet (-q)
--non-recursive (-N)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--ignore-externals
--config-dir DIR
```

Examples

Check out a working copy into a directory called `mine`:

```
$ svn checkout file:///tmp/repos/test mine
A  mine/a
A  mine/b
Checked out revision 2.
$ ls
mine
```

Check out two different directories into two separate working copies:

```
$ svn checkout file:///tmp/repos/test  file:///tmp/repos/quiz
A  test/a
```

```
A  test/b
Checked out revision 2.
A  quiz/l
A  quiz/m
Checked out revision 2.
$ ls
quiz  test
```

Check out two different directories into two separate working copies, but place both into a directory called `working-copies`:

```
$ svn checkout file:///tmp/repos/test  file:///tmp/repos/quiz working-copies
A  working-copies/test/a
A  working-copies/test/b
Checked out revision 2.
A  working-copies/quiz/l
A  working-copies/quiz/m
Checked out revision 2.
$ ls
working-copies
```

If you interrupt a checkout (or something else interrupts your checkout, like loss of connectivity, etc.), you can restart it either by issuing the identical checkout command again, or by updating the incomplete working copy:

```
$ svn checkout file:///tmp/repos/test test
A  test/a
A  test/b
^C
svn: The operation was interrupted
svn: caught SIGINT

$ svn checkout file:///tmp/repos/test test
A  test/c
A  test/d
^C
svn: The operation was interrupted
svn: caught SIGINT

$ cd test
$ svn update
A  test/e
A  test/f
Updated to revision 3.
```

Name

svn cleanup — Recursively clean up the working copy.

Synopsis

```
svn cleanup [PATH...]
```

Description

Recursively clean up the working copy, removing working copy locks and resuming unfinished operations. If you ever get a “working copy locked” error, run this command to remove stale locks and get your working copy into a usable state again.

If, for some reason, an **svn update** fails due to a problem running an external diff program (e.g. user input or network failure), pass the `--diff3-cmd` to allow cleanup to complete any merging with your external diff program. You can also specify any configuration directory with the `--config-dir` option, but you should need these options extremely infrequently.

Alternate Names

None

Changes

Working copy

Accesses Repository

No

Options

```
--diff3-cmd CMD
--config-dir DIR
```

Examples

Well, there's not much to the examples here as **svn cleanup** generates no output. If you pass no *PATH*, “`.`” is used.

```
$ svn cleanup

$ svn cleanup /path/to/working-copy
```

Name

svn commit — Send changes from your working copy to the repository.

Synopsis

```
svn commit [PATH...]
```

Description

Send changes from your working copy to the repository. If you do not supply a log message with your commit by using either the `--file` or `--message` option, **svn** will launch your editor for you to compose a commit message. See the `editor-cmd` section in the section called “Config”.

svn commit will send any lock tokens that it finds and will release locks on all *PATHS* committed (recursively), unless `--no-unlock` is passed.



If you begin a commit and Subversion launches your editor to compose the commit message, you can still abort without committing your changes. If you want to cancel your commit, just quit your editor without saving your commit message and Subversion will prompt you to either abort the commit, continue with no message, or edit the message again.

Alternate Names

ci (short for “check in”; not “co”, which is short for “checkout”)

Changes

Working copy, repository

Accesses Repository

Yes

Options

```
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--no-unlock
--non-recursive (-N)
--targets FILENAME
--force-log
--username USER
--password PASS
--no-auth-cache
--non-interactive
--encoding ENC
--config-dir DIR
```

Examples

Commit a simple modification to a file with the commit message on the command line and an implicit target of your current directory (“.”):

```
$ svn commit -m "added howto section."
Sending        a
Transmitting file data .
Committed revision 3.
```

Commit a modification to the file `foo.c` (explicitly specified on the command line) with the commit message in a file named `msg`:

```
$ svn commit -F msg foo.c
Sending        foo.c
Transmitting file data .
Committed revision 5.
```

If you want to use a file that's under version control for your commit message with `--file`, you need to pass the `--force-log` option:

```
$ svn commit --file file_under_vc.txt foo.c
svn: The log message file is under version control
svn: Log message file is a versioned file; use '--force-log' to override

$ svn commit --force-log --file file_under_vc.txt foo.c
Sending        foo.c
Transmitting file data .
Committed revision 6.
```

To commit a file scheduled for deletion:

```
$ svn commit -m "removed file 'c'."
Deleting       c

Committed revision 7.
```

Name

svn copy — Copy a file or directory in a working copy or in the repository.

Synopsis

```
svn copy SRC DST
```

Description

Copy a file in a working copy or in the repository. *SRC* and *DST* can each be either a working copy (WC) path or URL:

WC -> WC
: Copy and schedule an item for addition (with history).

WC -> URL
: Immediately commit a copy of WC to URL.

URL -> WC
: Check out URL into WC, and schedule it for addition.

URL -> URL
: Complete server-side copy. This is usually used to branch and tag.



You can only copy files within a single repository. Subversion does not support cross-repository copying.

Alternate Names

cp

Changes

Repository if destination is a URL.

Working copy if destination is a WC path.

Accesses Repository

If source or destination is in the repository, or if needed to look up the source revision number.

Options

```
--message (-m) TEXT
--file (-F) FILE
--revision (-r) REV
--quiet (-q)
--username USER
--password PASS
--no-auth-cache
--non-interactive
```

```
--force-log
--editor-cmd EDITOR
--encoding ENC
--config-dir DIR
```

Examples

Copy an item within your working copy (just schedules the copy—nothing goes into the repository until you commit):

```
$ svn copy foo.txt bar.txt
A         bar.txt
$ svn status
A  +   bar.txt
```

Copy an item in your working copy to a URL in the repository (an immediate commit, so you must supply a commit message):

```
$ svn copy near.txt file:///tmp/repos/test/far-away.txt -m "Remote copy."

Committed revision 8.
```

Copy an item from the repository to your working copy (just schedules the copy—nothing goes into the repository until you commit):



This is the recommended way to resurrect a dead file in your repository!

```
$ svn copy file:///tmp/repos/test/far-away near-here
A         near-here
```

And finally, copying between two URLs:

```
$ svn copy file:///tmp/repos/test/far-away file:///tmp/repos/test/over-there -m "r

Committed revision 9.
```



This is the easiest way to “tag” a revision in your repository—just **svn copy** that revision (usually HEAD) into your tags directory.

```
$ svn copy file:///tmp/repos/test/trunk file:///tmp/repos/test/tags/0.6.32-prerele

Committed revision 12.
```

And don't worry if you forgot to tag—you can always specify an older revision and tag anytime:

```
$ svn copy -r 11 file:///tmp/repos/test/trunk file:///tmp/repos/test/tags/0.6.32-p
```

```
Committed revision 13.
```

Name

svn delete — Delete an item from a working copy or the repository.

Synopsis

```
svn delete PATH...

svn delete URL...
```

Description

Items specified by *PATH* are scheduled for deletion upon the next commit. Files (and directories that have not been committed) are immediately removed from the working copy. The command will not remove any unversioned or modified items; use the `--force` option to override this behavior.

Items specified by URL are deleted from the repository via an immediate commit. Multiple URLs are committed atomically.

Alternate Names

del, remove, rm

Changes

Working copy if operating on files, repository if operating on URLs

Accesses Repository

Only if operating on URLs

Options

```
--force
--force-log
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--targets FILENAME
--username USER
--password PASS
--no-auth-cache
--non-interactive
--editor-cmd EDITOR
--encoding ENC
--config-dir DIR
```

Examples

Using **svn** to delete a file from your working copy deletes your local copy of the file, but merely schedules it to be deleted from the repository. When you commit, the file is deleted in the repository.

```
$ svn delete myfile
D         myfile

$ svn commit -m "Deleted file 'myfile'."
Deleting       myfile
Transmitting file data .
Committed revision 14.
```

Deleting a URL, however, is immediate, so you have to supply a log message:

```
$ svn delete -m "Deleting file 'yourfile'" file:///tmp/repos/test/yourfile

Committed revision 15.
```

Here's an example of how to force deletion of a file that has local mods:

```
$ svn delete over-there
svn: Attempting restricted operation for modified resource
svn: Use --force to override this restriction
svn: 'over-there' has local modifications

$ svn delete --force over-there
D         over-there
```

Name

svn diff — Display the differences between two revisions or paths.

Synopsis

```
diff [-c M | -r N[:M]] [TARGET[@REV]...]

diff [-r N[:M]] --old=OLD-TGT[@OLDREV] [--new=NEW-TGT[@NEWREV]] [PATH...]

diff OLD-URL[@OLDREV] NEW-URL[@NEWREV]
```

Description

Display the differences between two paths. The ways you can use **svn diff** are:

Use just **svn diff'**to display local modifications in a working copy.

Display the changes made to *TARGET*s as they are seen in *REV* between two revisions. *TARGET*s may be all working copy paths or all *URL*s. If *TARGET*s are working copy paths, *N* defaults to BASE and *M* to the working copy; if *URL*s, *N* must be specified and *M* defaults to HEAD. The "-c M" option is equivalent to "-r N:M" where `N = M-1`. Using "-c -M" does the reverse: "-r M:N" where `N = M-1`.

Display the differences between *OLD-TGT* as it was seen in *OLDREV* and *NEW-TGT* as it was seen ain *NEWREV*. *PATH*s, if given, are relative to *OLD-TGT* and *NEW-TGT* and restrict the output to differences for those paths. *OLD-TGT* and *NEW-TGT* may be working copy paths or *URL[@REV]*. *NEW-TGT* defaults to *OLD-TGT* if not specified. "-r N" makes OLDREV default to N, -r N:M makes *OLDREV* default to *N* and *NEWREV* default to *M*.

Shorthand for **svn diff --old=OLD-URL[@OLDREV] --new=NEW-URL[@NEWREV]**

svn diff -r N:M URL is shorthand for **svn diff -r N:M --old=URL --new=URL**.

svn diff [-r N[:M]] URL1[@N] URL2[@M] is shorthand for **svn diff [-r N[:M]] --old=URL1 --new=URL2**.

If *TARGET* is a URL, then revs N and M can be given either via the `--revision` or by using "@" notation as described earlier.

If *TARGET* is a working copy path, then the `--revision` option means:

`--revision N:M`
: The server compares *TARGET@N* and *TARGET@M*.

`--revision N`
: The client compares *TARGET@N* against working copy.

(no `--revision`)
: The client compares base and working copies of *TARGET*.

If the alternate syntax is used, the server compares *URL1* and *URL2* at revisions *N* and *M* respectively. If either *N* or *M* are omitted, a value of HEAD is assumed.

By default, **svn diff** ignores the ancestry of files and merely compares the contents of the two files being compared. If you use `--notice-ancestry`, the ancestry of the paths in question will be taken into consideration when comparing revisions (that is, if you run **svn diff** on two files with identical contents but different ancestry you will see the entire contents of the file as having been removed and added again).

Alternate Names

di

Changes

Nothing

Accesses Repository

For obtaining differences against anything but BASE revision in your working copy

Options

```
--revision (-r) ARG
--change (-c) ARG
--old ARG
--new ARG
--non-recursive (-N)
--diff-cmd CMD
--extensions (-x) "ARGS"
--no-diff-deleted
--notice-ancestry
--summarize
--force
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

Compare BASE and your working copy (one of the most popular uses of **svn diff**):

```
$ svn diff COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 4404)
+++ COMMITTERS (working copy)
```

See what changed in the file COMMITTERS revision 9115:

```
$ svn diff -c 9115 COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3900)
+++ COMMITTERS (working copy)
```

See how your working copy's modifications compare against an older revision:

```
$ svn diff -r 3900 COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3900)
+++ COMMITTERS (working copy)
```

Compare revision 3000 to revision 3500 using “@” syntax:

```
$ svn diff http://svn.collab.net/repos/svn/trunk/COMMITTERS@3000 http://svn.collab
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
…
```

Compare revision 3000 to revision 3500 using range notation (you only pass the one URL in this case):

```
$ svn diff -r 3000:3500 http://svn.collab.net/repos/svn/trunk/COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
```

Compare revision 3000 to revision 3500 of all files in `trunk` using range notation:

```
$ svn diff -r 3000:3500 http://svn.collab.net/repos/svn/trunk
```

Compare revision 3000 to revision 3500 of only three files in `trunk` using range notation:

```
$ svn diff -r 3000:3500 --old http://svn.collab.net/repos/svn/trunk COMMITTERS REA
```

If you have a working copy, you can obtain the differences without typing in the long URLs:

```
$ svn diff -r 3000:3500 COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
```

Use `--diff-cmd` *`CMD`* `-x` to pass arguments directly to the external diff program

```
$ svn diff --diff-cmd /usr/bin/diff -x "-i -b" COMMITTERS
Index: COMMITTERS
===================================================================
0a1,2
```

```
> This is a test
>
```

Name

svn export — Export a clean directory tree.

Synopsis

```
svn export [-r REV] URL[@PEGREV] [PATH]

svn export [-r REV] PATH1[@PEGREV] [PATH2]
```

Description

The first form exports a clean directory tree from the repository specified by URL, at revision *REV* if it is given, otherwise at HEAD, into *PATH*. If *PATH* is omitted, the last component of the *URL* is used for the local directory name.

The second form exports a clean directory tree from the working copy specified by *PATH1* into *PATH2*. All local changes will be preserved, but files not under version control will not be copied.

Alternate Names

None

Changes

Local disk

Accesses Repository

Only if exporting from a URL

Options

```
--revision (-r) REV
--quiet (-q)
--force
--username USER
--password PASS
--no-auth-cache
--non-interactive
--non-recursive (-N)
--config-dir DIR
--native-eol EOL
--ignore-externals
```

Examples

Export from your working copy (doesn't print every file and directory):

```
$ svn export a-wc my-export
Export complete.
```

Export directly from the repository (prints every file and directory):

```
$ svn export file:///tmp/repos my-export
A  my-export/test
A  my-export/quiz
…
Exported revision 15.
```

When rolling operating-system-specific release packages, it can be useful to export a tree which uses a specific EOL character for line endings. The `--native-eol` option will do this, but it only affects files that have `svn:eol-style = native` properties attached to them. For example, to export a tree with all CRLF line endings (possibly for a Windows .zip file distribution):

```
$ svn export file:///tmp/repos my-export --native-eol CRLF
A  my-export/test
A  my-export/quiz
…
Exported revision 15.
```

You can specify `LR`, `CR`, or `CRLF` as a line ending type with the `--native-eol` option.

Name

svn help — Help!

Synopsis

```
svn help [SUBCOMMAND...]
```

Description

This is your best friend when you're using Subversion and this book isn't within reach!

Alternate Names

?, h

The options `-?`, `-h` and `--help` have the same effect as using the **help** subcommand.

Changes

Nothing

Accesses Repository

No

Options

```
--config-dir DIR
```

Name

svn import — Commit an unversioned file or tree into the repository.

Synopsis

```
svn import [PATH] URL
```

Description

Recursively commit a copy of *PATH* to *URL*. If *PATH* is omitted "." is assumed. Parent directories are created in the repository as necessary.

Alternate Names

None

Changes

Repository

Accesses Repository

Yes

Options

```
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--non-recursive (-N)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--force-log
--editor-cmd EDITOR
--encoding ENC
--config-dir DIR
--auto-props
--no-auto-props
--ignore-externals
```

Examples

This imports the local directory `myproj` into `trunk/misc` in your repository. The directory `trunk/misc` need not exist before you import into it—**svn import** will recursively create directories for you.

```
$ svn import -m "New import" myproj http://svn.red-bean.com/repos/trunk/misc
Adding         myproj/sample.txt
…
Transmitting file data .........
Committed revision 16.
```

Be aware that this will *not* create a directory named `myproj` in the repository. If that's what you want, simply add `myproj` to the end of the URL:

```
$ svn import -m "New import" myproj http://svn.red-bean.com/repos/trunk/misc/mypro
Adding         myproj/sample.txt
…
Transmitting file data .........
Committed revision 16.
```

After importing data, note that the original tree is *not* under version control. To start working, you still need to **svn checkout** a fresh working copy of the tree.

Name

svn info — Display information about a local or remote item.

Synopsis

```
svn info [TARGET[@REV]...]
```

Description

Print information about the working copy paths or URLs specified. The information shown for both may include:

- Path
- Name
- URL
- Repository Root
- Repository UUID
- Revision
- Node Kind
- Last Changed Author
- Last Changed Revision
- Last Changed Date
- Lock Token
- Lock Owner
- Lock Created (date)
- Lock Expires (date)

Additional kinds of information available only for working copy paths are:

- Schedule
- Copied From URL
- Copied From Rev
- Text Last Updated
- Properties Last Updated
- Checksum

- Conflict Previous Base File
- Conflict Previous Working File
- Conflict Current Base File
- Conflict Properties File

Alternate Names

None

Changes

Nothing

Accesses Repository

Only if operating on URLs

Options

```
--revision (-r) REV
--recursive (-R)
--targets FILENAME
--incremental
--xml
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

svn info will show you all the useful information that it has for items in your working copy. It will show information for files:

```
$ svn info foo.c
Path: foo.c
Name: foo.c
URL: http://svn.red-bean.com/repos/test/foo.c
Repository Root: http://svn.red-bean.com/repos/test
Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25
Revision: 4417
Node Kind: file
Schedule: normal
Last Changed Author: sally
Last Changed Rev: 20
Last Changed Date: 2003-01-13 16:43:13 -0600 (Mon, 13 Jan 2003)
Text Last Updated: 2003-01-16 21:18:16 -0600 (Thu, 16 Jan 2003)
Properties Last Updated: 2003-01-13 21:50:19 -0600 (Mon, 13 Jan 2003)
Checksum: d6aeb60b0662ccceb6bce4bac344cb66
```

It will also show information for directories:

```
$ svn info vendors
Path: vendors
URL: http://svn.red-bean.com/repos/test/vendors
Repository Root: http://svn.red-bean.com/repos/test
Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25
Revision: 19
Node Kind: directory
Schedule: normal
Last Changed Author: harry
Last Changed Rev: 19
Last Changed Date: 2003-01-16 23:21:19 -0600 (Thu, 16 Jan 2003)
Properties Last Updated: 2003-01-16 23:39:02 -0600 (Thu, 16 Jan 2003)
```

svn info also acts on URLs (also note that the file readme.doc in this example is locked, so lock information is also provided):

```
$ svn info http://svn.red-bean.com/repos/test/readme.doc
Path: readme.doc
Name: readme.doc
URL: http://svn.red-bean.com/repos/test/readme.doc
Repository Root: http://svn.red-bean.com/repos/test
Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25
Revision: 1
Node Kind: file
Schedule: normal
Last Changed Author: sally
Last Changed Rev: 42
Last Changed Date: 2003-01-14 23:21:19 -0600 (Tue, 14 Jan 2003)
Lock Token: opaquelocktoken:14011d4b-54fb-0310-8541-dbd16bd471b2
Lock Owner: harry
Lock Created: 2003-01-15 17:35:12 -0600 (Wed, 15 Jan 2003)
Lock Comment (1 line):
My test lock comment
```

Name

svn list — List directory entries in the repository.

Synopsis

```
svn list [TARGET[@REV]...]
```

Description

List each *TARGET* file and the contents of each *TARGET* directory as they exist in the repository. If *TARGET* is a working copy path, the corresponding repository URL will be used.

The default *TARGET* is "`.`", meaning the repository URL of the current working copy directory.

With `--verbose`, **svn list** shows the following fields for each item:

- Revision number of the last commit
- Author of the last commit
- If locked, the letter "O" (See svn info for details).
- Size (in bytes)
- Date and time of the last commit

With `--xml`, output is in XML format (with a header and an enclosing document element unless `--incremental` is also specified). All of the information is present; the `--verbose` option is not accepted.

Alternate Names

ls

Changes

Nothing

Accesses Repository

Yes

Options

```
--revision (-r) REV
--verbose (-v)
--recursive (-R)
--incremental
--xml
--username USER
--password PASS
--no-auth-cache
--non-interactive
```

```
--config-dir DIR
```

Examples

svn list is most useful if you want to see what files a repository has without downloading a working copy:

```
$ svn list http://svn.red-bean.com/repos/test/support
README.txt
INSTALL
examples/
…
```

You can pass the `--verbose` option for additional information, rather like the UNIX command **ls -l**:

```
$ svn list --verbose file:///tmp/repos
     16 sally         28361 Jan 16 23:18 README.txt
     27 sally             0 Jan 18 15:27 INSTALL
     24 harry               Jan 18 11:27 examples/
```

For further details, see the section called “**svn list**”.

Name

svn lock — Lock working copy paths or URLs in the repository, so that no other user can commit changes to them.

Synopsis

```
svn lock TARGET...
```

Description

Lock each *TARGET*. If any *TARGET* is already locked by another user, print a warning and continue locking the rest of the *TARGET*s. Use `--force` to steal a lock from another user or working copy.

Alternate Names

None

Changes

Working Copy, Repository

Accesses Repository

Yes

Options

```
--targets FILENAME
--message (-m) TEXT
--file (-F) FILE
--force-log
--encoding ENC
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
--force
```

Examples

Lock two files in your working copy:

```
$ svn lock tree.jpg house.jpg
'tree.jpg' locked by user 'harry'.
'house.jpg' locked by user 'harry'.
```

Lock a file in your working copy that is currently locked by another user:

```
$ svn lock tree.jpg
```

```
svn: warning: Path '/tree.jpg is already locked by user 'sally in \
     filesystem '/svn/repos/db'

$ svn lock --force tree.jpg
'tree.jpg' locked by user 'harry'.
```

Lock a file without a working copy:

```
$ svn lock http://svn.red-bean.com/repos/test/tree.jpg
'tree.jpg' locked by user 'harry'.
```

For further details, see the section called “Locking”.

Name

svn log — Display commit log messages.

Synopsis

```
svn log [PATH]

svn log URL [PATH...]

svn log URL[@REV] [PATH...]
```

Description

Shows log messages from the repository. If no arguments are supplied, **svn log** shows the log messages for all files and directories inside of (and including) the current working directory of your working copy. You can refine the results by specifying a path, one or more revisions, or any combination of the two. The default revision range for a local path is `BASE:1`.

If you specify a URL alone, then it prints log messages for everything that the URL contains. If you add paths past the URL, only messages for those paths under that URL will be printed. The default revision range for a URL is `HEAD:1`.

With `--verbose`, **svn log** will also print all affected paths with each log message. With `--quiet`, **svn log** will not print the log message body itself (this is compatible with `--verbose`).

Each log message is printed just once, even if more than one of the affected paths for that revision were explicitly requested. Logs follow copy history by default. Use `--stop-on-copy` to disable this behavior, which can be useful for determining branch points.

Alternate Names

None

Changes

Nothing

Accesses Repository

Yes

Options

```
--revision (-r) REV
--quiet (-q)
--verbose (-v)
--targets FILENAME
--stop-on-copy
--incremental
--limit NUM
--xml
--username USER
```

```
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

You can see the log messages for all the paths that changed in your working copy by running **svn log** from the top:

```
$ svn log
------------------------------------------------------------------------
r20 | harry | 2003-01-17 22:56:19 -0600 (Fri, 17 Jan 2003) | 1 line

Tweak.
------------------------------------------------------------------------
r17 | sally | 2003-01-16 23:21:19 -0600 (Thu, 16 Jan 2003) | 2 lines
…
```

Examine all log messages for a particular file in your working copy:

```
$ svn log foo.c
------------------------------------------------------------------------
r32 | sally | 2003-01-13 00:43:13 -0600 (Mon, 13 Jan 2003) | 1 line

Added defines.
------------------------------------------------------------------------
r28 | sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines
…
```

If you don't have a working copy handy, you can log a URL:

```
$ svn log http://svn.red-bean.com/repos/test/foo.c
------------------------------------------------------------------------
r32 | sally | 2003-01-13 00:43:13 -0600 (Mon, 13 Jan 2003) | 1 line

Added defines.
------------------------------------------------------------------------
r28 | sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines
…
```

If you want several distinct paths underneath the same URL, you can use the `URL [PATH...]` syntax.

```
$ svn log http://svn.red-bean.com/repos/test/ foo.c bar.c
------------------------------------------------------------------------
r32 | sally | 2003-01-13 00:43:13 -0600 (Mon, 13 Jan 2003) | 1 line

Added defines.
------------------------------------------------------------------------
r31 | harry | 2003-01-10 12:25:08 -0600 (Fri, 10 Jan 2003) | 1 line

Added new file bar.c
------------------------------------------------------------------------
r28 | sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines
```

…

When you're concatenating the results of multiple calls to the log command, you may want to use the `--incremental` option. **svn log** normally prints out a dashed line at the beginning of a log message, after each subsequent log message, and following the final log message. If you ran **svn log** on a range of two revisions, you would get this:

```
$ svn log -r 14:15
------------------------------------------------------------------------
r14 | …

------------------------------------------------------------------------
r15 | …

------------------------------------------------------------------------
```

However, if you wanted to gather 2 non-sequential log messages into a file, you might do something like this:

```
$ svn log -r 14 > mylog
$ svn log -r 19 >> mylog
$ svn log -r 27 >> mylog
$ cat mylog
------------------------------------------------------------------------
r14 | …

------------------------------------------------------------------------
------------------------------------------------------------------------
r19 | …

------------------------------------------------------------------------
------------------------------------------------------------------------
r27 | …

------------------------------------------------------------------------
```

You can avoid the clutter of the double dashed lines in your output by using the incremental option:

```
$ svn log --incremental -r 14 > mylog
$ svn log --incremental -r 19 >> mylog
$ svn log --incremental -r 27 >> mylog
$ cat mylog
------------------------------------------------------------------------
r14 | …

------------------------------------------------------------------------
r19 | …

------------------------------------------------------------------------
r27 | …
```

The `--incremental` option provides similar output control when using the `--xml` option.



If you run **svn log** on a specific path and provide a specific revision and get no output at all

```
$ svn log -r 20 http://svn.red-bean.com/untouched.txt
------------------------------------------------------------------------
```

That just means that the path was not modified in that revision. If you log from the top of the repository, or know the file that changed in that revision, you can specify it explicitly:

```
$ svn log -r 20 touched.txt
------------------------------------------------------------------------
r20 | sally | 2003-01-17 22:56:19 -0600 (Fri, 17 Jan 2003) | 1 line

Made a change.
------------------------------------------------------------------------
```

Name

svn merge — Apply the differences between two sources to a working copy path.

Synopsis

```
svn merge [-c M | -r N:M] SOURCE[@REV] [WCPATH]

svn merge sourceURL1[@N] sourceURL2[@M] [WCPATH]

svn merge sourceWCPATH1@N sourceWCPATH2@M [WCPATH]
```

Description

In the first and second forms, the source paths (URLs in the first form, working copy paths in the second) are specified at revisions *N* and *M*. These are the two sources to be compared. The revisions default to HEAD if omitted.

The `-c M` option is equivalent to `-r N:M` where *N = M-1*. Using `-c -M` does the reverse: `-r M:N` where *N = M-1*.

In the third form, *SOURCE* can be a URL or working copy item, in which case the corresponding URL is used. This URL, at revisions *N* and *M*, defines the two sources to be compared.

WCPATH is the working copy path that will receive the changes. If *WCPATH* is omitted, a default value of "`.`" is assumed, unless the sources have identical basenames that match a file within "`.`": in which case, the differences will be applied to that file.

Unlike **svn diff**, the merge command takes the ancestry of a file into consideration when performing a merge operation. This is very important when you're merging changes from one branch into another and you've renamed a file on one branch but not the other.

Alternate Names

None

Changes

Working copy

Accesses Repository

Only if working with URLs

Options

```
--revision (-r) REV
--change (-c) REV
--non-recursive (-N)
--quiet (-q)
--force
--dry-run
--diff3-cmd CMD
--extensions (-x) ARG
```

```
--ignore-ancestry
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

Merge a branch back into the trunk (assuming that you have a working copy of the trunk, and that the branch was created in revision 250):

```
$ svn merge -r 250:HEAD http://svn.red-bean.com/repos/branches/my-branch
U  myproj/tiny.txt
U  myproj/thhgttg.txt
U  myproj/win.txt
U  myproj/flo.txt
```

If you branched at revision 23, and you want to merge changes on trunk into your branch, you could do this from inside the working copy of your branch:

```
$ svn merge -r 23:30 file:///tmp/repos/trunk/vendors
U  myproj/thhgttg.txt
…
```

To merge changes to a single file:

```
$ cd myproj
$ svn merge -r 30:31 thhgttg.txt
U  thhgttg.txt
```

Name

svn mkdir — Create a new directory under version control.

Synopsis

```
svn mkdir PATH...

svn mkdir URL...
```

Description

Create a directory with a name given by the final component of the *PATH* or URL. A directory specified by a working copy *PATH* is scheduled for addition in the working copy. A directory specified by a URL is created in the repository via an immediate commit. Multiple directory URLs are committed atomically. In both cases all the intermediate directories must already exist.

Alternate Names

None

Changes

Working copy, repository if operating on a URL

Accesses Repository

Only if operating on a URL

Options

```
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--editor-cmd EDITOR
--encoding ENC
--force-log
--config-dir DIR
```

Examples

Create a directory in your working copy:

```
$ svn mkdir newdir
A         newdir
```

Create one in the repository (instant commit, so a log message is required):

```
$ svn mkdir -m "Making a new dir." http://svn.red-bean.com/repos/newdir
Committed revision 26.
```

Name

svn move — Move a file or directory.

Synopsis

```
svn move SRC DST
```

Description

This command moves a file or directory in your working copy or in the repository.



This command is equivalent to an **svn copy** followed by **svn delete**.



Subversion does not support moving between working copies and URLs. In addition, you can only move files within a single repository—Subversion does not support cross-repository moving.

WC -> WC
: Move and schedule a file or directory for addition (with history).

URL -> URL
: Complete server-side rename.

Alternate Names

mv, rename, ren

Changes

Working copy, repository if operating on a URL

Accesses Repository

Only if operating on a URL

Options

```
--message (-m) TEXT
--file (-F) FILE
--revision (-r) REV (Deprecated)
--quiet (-q)
--force
--username USER
--password PASS
--no-auth-cache
--non-interactive
--editor-cmd EDITOR
--encoding ENC
--force-log
--config-dir DIR
```

Examples

Move a file in your working copy:

```
$ svn move foo.c bar.c
A         bar.c
D         foo.c
```

Move a file in the repository (an immediate commit, so it requires a commit message):

```
$ svn move -m "Move a file" http://svn.red-bean.com/repos/foo.c \
                            http://svn.red-bean.com/repos/bar.c

Committed revision 27.
```

Name

svn propdel — Remove a property from an item.

Synopsis

```
svn propdel PROPNAME [PATH...]

svn propdel PROPNAME --revprop -r REV [TARGET]
```

Description

This removes properties from files, directories, or revisions. The first form removes versioned properties in your working copy, while the second removes unversioned remote properties on a repository revision (*TARGET* only determines which repository to access).

Alternate Names

pdel, pd

Changes

Working copy, repository only if operating on a URL

Accesses Repository

Only if operating on a URL

Options

```
--quiet (-q)
--recursive (-R)
--revision (-r) REV
--revprop
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

Delete a property from a file in your working copy

```
$ svn propdel svn:mime-type  some-script
property 'svn:mime-type' deleted from 'some-script'.
```

Delete a revision property:

```
$ svn propdel --revprop -r 26 release-date
property 'release-date' deleted from repository revision '26'
```

Name

svn propedit — Edit the property of one or more items under version control.

Synopsis

```
svn propedit PROPNAME PATH...

svn propedit PROPNAME --revprop -r REV [TARGET]
```

Description

Edit one or more properties using your favorite editor. The first form edits versioned properties in your working copy, while the second edits unversioned remote properties on a repository revision (*TARGET* only determines which repository to access).

Alternate Names

pedit, pe

Changes

Working copy, repository only if operating on a URL

Accesses Repository

Only if operating on a URL

Options

```
--revision (-r) REV
--revprop
--username USER
--password PASS
--no-auth-cache
--non-interactive
--encoding ENC
--editor-cmd EDITOR
--config-dir DIR
```

Examples

svn propedit makes it easy to modify properties that have multiple values:

```
$ svn propedit svn:keywords  foo.c
    <svn will launch your favorite editor here, with a buffer open
    containing the current contents of the svn:keywords property.  You
    can add multiple values to a property easily here by entering one
    value per line.>
Set new value for property 'svn:keywords' on 'foo.c'
```

Name

svn propget — Print the value of a property.

Synopsis

```
svn propget PROPNAME [TARGET[@REV]...]

svn propget PROPNAME --revprop -r REV [URL]
```

Description

Print the value of a property on files, directories, or revisions. The first form prints the versioned property of an item or items in your working copy, while the second prints unversioned remote property on a repository revision. See the section called “Properties” for more information on properties.

Alternate Names

pget, pg

Changes

Working copy, repository only if operating on a URL

Accesses Repository

Only if operating on a URL

Options

```
--recursive (-R)
--revision (-r) REV
--revprop
--strict
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

Examine a property of a file in your working copy:

```
$ svn propget svn:keywords foo.c
Author
Date
Rev
```

The same goes for a revision property:

```
$ svn propget svn:log --revprop -r 20
Began journal.
```

Name

svn proplist — List all properties.

Synopsis

```
svn proplist [TARGET[@REV]...]

svn proplist --revprop -r REV [TARGET]
```

Description

List all properties on files, directories, or revisions. The first form lists versioned properties in your working copy, while the second lists unversioned remote properties on a repository revision (*TARGET* only determines which repository to access).

Alternate Names

plist, pl

Changes

Working copy, repository only if operating on a URL

Accesses Repository

Only if operating on a URL

Options

```
--verbose (-v)
--recursive (-R)
--revision (-r) REV
--quiet (-q)
--revprop
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

You can use proplist to see the properties on an item in your working copy:

```
$ svn proplist foo.c
Properties on 'foo.c':
  svn:mime-type
  svn:keywords
  owner
```

But with the `--verbose` flag, svn proplist is extremely handy as it also shows you the values for the properties:

```
$ svn proplist --verbose foo.c
Properties on 'foo.c':
  svn:mime-type : text/plain
  svn:keywords : Author Date Rev
  owner : sally
```

Name

svn propset — Set PROPNAME to PROPVAL on files, directories, or revisions.

Synopsis

```
svn propset PROPNAME [PROPVAL | -F VALFILE] PATH...

svn propset PROPNAME --revprop -r REV [PROPVAL | -F VALFILE] [TARGET]
```

Description

Set *PROPNAME* to *PROPVAL* on files, directories, or revisions. The first example creates a versioned, local property change in the working copy, and the second creates an unversioned, remote property change on a repository revision (*TARGET* only determines which repository to access).



Subversion has a number of “special” properties that affect its behavior. See the section called “Subversion properties” for more on these properties.

Alternate Names

pset, ps

Changes

Working copy, repository only if operating on a URL

Accesses Repository

Only if operating on a URL

Options

```
--file (-F) FILE
--quiet (-q)
--revision (-r) REV
--targets FILENAME
--recursive (-R)
--revprop
--username USER
--password PASS
--no-auth-cache
--non-interactive
--encoding ENC
--force
--config-dir DIR
```

Examples

Set the mime type on a file:

```
$ svn propset svn:mime-type image/jpeg foo.jpg
property 'svn:mime-type' set on 'foo.jpg'
```

On a UNIX system, if you want a file to have the executable permission set:

```
$ svn propset svn:executable ON somescript
property 'svn:executable' set on 'somescript'
```

Perhaps you have an internal policy to set certain properties for the benefit of your coworkers:

```
$ svn propset owner sally foo.c
property 'owner' set on 'foo.c'
```

If you made a mistake in a log message for a particular revision and want to change it, use `--revprop` and set `svn:log` to the new log message:

```
$ svn propset --revprop -r 25 svn:log "Journaled about trip to New York."
property 'svn:log' set on repository revision '25'
```

Or, if you don't have a working copy, you can provide a URL.

```
$ svn propset --revprop -r 26 svn:log "Document nap." http://svn.red-bean.com/repo
property 'svn:log' set on repository revision '25'
```

Lastly, you can tell propset to take its input from a file. You could even use this to set the contents of a property to something binary:

```
$ svn propset owner-pic -F sally.jpg moo.c
property 'owner-pic' set on 'moo.c'
```



By default, you cannot modify revision properties in a Subversion repository. Your repository administrator must explicitly enable revision property modifications by creating a hook named `pre-revprop-change`. See the section called “Implementing Repository Hooks” for more information on hook scripts.

Name

svn resolved — Remove “conflicted” state on working copy files or directories.

Synopsis

```
svn resolved PATH...
```

Description

Remove “conflicted” state on working copy files or directories. This routine does not semantically resolve conflict markers; it merely removes conflict-related artifact files and allows *PATH* to be committed again; that is, it tells Subversion that the conflicts have been “resolved”. See the section called “Resolve Conflicts (Merging Others' Changes)” for an in-depth look at resolving conflicts.

Alternate Names

None

Changes

Working copy

Accesses Repository

No

Options

```
--targets FILENAME
--recursive (-R)
--quiet (-q)
--config-dir DIR
```

Examples

If you get a conflict on an update, your working copy will sprout three new files:

```
$ svn update
C  foo.c
Updated to revision 31.
$ ls
foo.c
foo.c.mine
foo.c.r30
foo.c.r31
```

Once you've resolved the conflict and `foo.c` is ready to be committed, run **svn resolved** to let your working copy know you've taken care of everything.



You *can* just remove the conflict files and commit, but **svn resolved** fixes up some bookkeeping data in the working copy administrative area in addition to removing the conflict files, so we recommend that you use this command.

Name

svn revert — Undo all local edits.

Synopsis

```
svn revert PATH...
```

Description

Reverts any local changes to a file or directory and resolves any conflicted states. **svn revert** will not only revert the contents of an item in your working copy, but also any property changes. Finally, you can use it to undo any scheduling operations that you may have done (e.g. files scheduled for addition or deletion can be “unscheduled”).

Alternate Names

None

Changes

Working copy

Accesses Repository

No

Options

```
--targets FILENAME
--recursive (-R)
--quiet (-q)
--config-dir DIR
```

Examples

Discard changes to a file:

```
$ svn revert foo.c
Reverted foo.c
```

If you want to revert a whole directory of files, use the `--recursive` flag:

```
$ svn revert --recursive .
Reverted newdir/afile
Reverted foo.c
Reverted bar.txt
```

Lastly, you can undo any scheduling operations:

```
$ svn add mistake.txt whoops
```

```
A         mistake.txt
A         whoops
A         whoops/oopsie.c

$ svn revert mistake.txt whoops
Reverted mistake.txt
Reverted whoops

$ svn status
?      mistake.txt
?      whoops
```



svn revert is inherently dangerous, since its entire purpose is to throw away data—namely, your uncommitted changes. Once you've reverted, Subversion provides *no way* to get back those uncommitted changes.

If you provide no targets to **svn revert**, it will do nothing—to protect you from accidentally losing changes in your working copy, **svn revert** requires you to provide at least one target.

Name

svn status — Print the status of working copy files and directories.

Synopsis

```
svn status [PATH...]
```

Description

Print the status of working copy files and directories. With no arguments, it prints only locally modified items (no repository access). With `--show-updates`, it adds working revision and server out-of-date information. With `--verbose`, it prints full revision information on every item.

The first six columns in the output are each one character wide, and each column gives you information about different aspects of each working copy item.

The first column indicates that an item was added, deleted, or otherwise changed.

' '
: No modifications.

'A'
: Item is scheduled for Addition.

'D'
: Item is scheduled for Deletion.

'M'
: Item has been modified.

'R'
: Item has been replaced in your working copy. This means the file was scheduled for deletion, and then a new file with the same name was scheduled for addition in its place.

'C'
: The contents (as opposed to the properties) of the item conflict with updates received from the repository.

'X'
: Item is present because of an externals definition.

'I'
: Item is being ignored (e.g. with the `svn:ignore` property).

'?'
: Item is not under version control.

'!'
: Item is missing (e.g. you moved or deleted it without using **svn**). This also indicates that a directory is incomplete (a checkout or update was interrupted).

'~'
: Item is versioned as one kind of object (file, directory, link), but has been replaced by dif-

ferent kind of object.

The second column tells the status of a file's or directory's properties.

' '
No modifications.

'M'
Properties for this item have been modified.

'C'
Properties for this item are in conflict with property updates received from the repository.

The third column is populated only if the working copy directory is locked. (See the section called “Sometimes You Just Need to Clean Up”.)

' '
Item is not locked.

'L'
Item is locked.

The fourth column is populated only if the item is scheduled for addition-with-history.

' '
No history scheduled with commit.

'+'
History scheduled with commit.

The fifth column is populated only if the item is switched relative to its parent (see the section called “Traversing Branches”).

' '
Item is a child of its parent directory.

'S'
Item is switched.

The sixth column is populated with lock information.

' '
When `--show-updates` is used, the file is not locked. If `--show-updates` is *not* used, this merely means that the file is not locked in this working copy.

K
File is locked in this working copy.

O
File is locked either by another user or in another working copy. This only appears when -

`-show-updates` is used.

T
: File was locked in this working copy, but the lock has been “stolen” and is invalid. The file is currently locked in the repository. This only appears when `--show-updates` is used.

B
: File was locked in this working copy, but the lock has been “broken” and is invalid. The file is no longer locked This only appears when `--show-updates` is used.

The out-of-date information appears in the seventh column (only if you pass the `--show-updates` option).

' '
: The item in your working copy is up-to-date.

'*'
: A newer revision of the item exists on the server.

The remaining fields are variable width and delimited by spaces. The working revision is the next field if the `--show-updates` or `--verbose` options are passed.

If the `--verbose` option is passed, the last committed revision and last committed author are displayed next.

The working copy path is always the final field, so it can include spaces.

Alternate Names

stat, st

Changes

Nothing

Accesses Repository

Only if using `--show-updates`

Options

```
--show-updates (-u)
--verbose (-v)
--non-recursive (-N)
--quiet (-q)
--no-ignore
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
--ignore-externals
```

Examples

This is the easiest way to find out what changes you have made to your working copy:

```
$ svn status wc
 M     wc/bar.c
A  +   wc/qax.c
```

If you want to find out what files in your working copy are out-of-date, pass the `--show-updates` option (this will *not* make any changes to your working copy). Here you can see that `wc/foo.c` has changed in the repository since we last updated our working copy:

```
$ svn status --show-updates wc
 M           965    wc/bar.c
       *     965    wc/foo.c
A  +         965    wc/qax.c
Status against revision:    981
```



`--show-updates` *only* places an asterisk next to items that are out of date (that is, items that will be updated from the repository if you run **svn update**). `--show-updates` does *not* cause the status listing to reflect the repository's version of the item (although you can see the revision number in the repository by passing the `--verbose` option).

And finally, the most information you can get out of the status subcommand:

```
$ svn status --show-updates --verbose wc
 M           965       938 sally        wc/bar.c
       *     965       922 harry        wc/foo.c
A  +         965       687 harry        wc/qax.c
             965       687 harry        wc/zig.c
Head revision:   981
```

For many more examples of **svn status**, see the section called “See an overview of your changes”.

Name

svn switch — Update working copy to a different URL.

Synopsis

```
svn switch URL [PATH]

switch --relocate FROM TO [PATH...]
```

Description

The first variant of this subcommand (without the `--relocate` option) updates your working copy to point to a new URL—usually a URL which shares a common ancestor with your working copy, although not necessarily. This is the Subversion way to move a working copy to a new branch. See the section called “Traversing Branches” for an in-depth look at switching.

The `--relocate` option causes **svn switch** to do something different: it updates your working copy to point to *the same* repository directory, only at a different URL (typically because an administrator has moved the repository to another server, or to another URL on the same server).

Alternate Names

sw

Changes

Working copy

Accesses Repository

Yes

Options

```
--revision (-r) REV
--non-recursive (-N)
--quiet (-q)
--diff3-cmd CMD
--relocate FROM TO
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

If you're currently inside the directory `vendors`, which was branched to `vendors-with-fix`, and you'd like to switch your working copy to that branch:

```
$ svn switch http://svn.red-bean.com/repos/branches/vendors-with-fix .
U  myproj/foo.txt
```

```
U  myproj/bar.txt
U  myproj/baz.c
U  myproj/qux.c
Updated to revision 31.
```

And to switch back, just provide the URL to the location in the repository from which you originally checked out your working copy:

```
$ svn switch http://svn.red-bean.com/repos/trunk/vendors .
U  myproj/foo.txt
U  myproj/bar.txt
U  myproj/baz.c
U  myproj/qux.c
Updated to revision 31.
```



You can just switch part of your working copy to a branch if you don't want to switch your entire working copy.

Sometimes an administrator might change the “base location” of your repository—in other words, the contents of the repository doesn't change, but the main URL used to reach the root of the repository does. For example, the hostname may change, the URL scheme may change, or any part of the URL which leads to the repository itself may change. Rather than check out a new working copy, you can have the **svn switch** command “rewrite” the beginnings of all the URLs in your working copy. Use the `--relocate` option to do the substitution. No file contents are changed, nor is the repository contacted. It's similar to running a Perl script over your working copy `.svn/` directories which runs **s/OldRoot/NewRoot/**.

```
$ svn checkout file:///tmp/repos test
A  test/a
A  test/b
…

$ mv repos newlocation
$ cd test/

$ svn update
svn: Unable to open an ra_local session to URL
svn: Unable to open repository 'file:///tmp/repos'

$ svn switch --relocate file:///tmp/repos file:///tmp/newlocation .
$ svn update
At revision 3.
```



Be careful when using the `--relocate` option. If you mistype the argument, you might end up creating nonsensical URLs within your working copy that render the whole workspace unusable and tricky to fix. It's also important to understand exactly when one should or shouldn't use `--relocate`. Here's the rule of thumb:

- If the working copy needs to reflect a new directory *within* the repository, then use just **svn switch**.
- If the working copy still reflects the same repository directory, but the location of

the repository itself has changed, then use **svn switch --relocate**.

Name

svn unlock — Unlock working copy paths or URLs.

Synopsis

```
svn unlock TARGET...
```

Description

Unlock each *TARGET*. If any *TARGET* is either locked by another user or no valid lock token exists in the working copy, print a warning and continue unlocking the rest of the *TARGET*s. Use `--force` to break a lock belonging to another user or working copy.

Alternate Names

None

Changes

Working Copy, Repository

Accesses Repository

Yes

Options

```
--targets FILENAME
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
--force
```

Examples

Unlock two files in your working copy:

```
$ svn unlock tree.jpg house.jpg
'tree.jpg' unlocked.
'house.jpg' unlocked.
```

Unlock a file in your working copy that is currently locked by another user:

```
$ svn unlock tree.jpg
svn: 'tree.jpg' is not locked in this working copy
$ svn unlock --force tree.jpg
'tree.jpg' unlocked.
```

Unlock a file without a working copy:

```
$ svn unlock http://svn.red-bean.com/repos/test/tree.jpg
'tree.jpg unlocked.
```

For further details, see the section called “Locking”.

Name

svn update — Update your working copy.

Synopsis

```
svn update [PATH...]
```

Description

svn update brings changes from the repository into your working copy. If no revision is given, it brings your working copy up-to-date with the `HEAD` revision. Otherwise, it synchronizes the working copy to the revision given by the `--revision` option. As part of the synchronization, **svn update** also removes any stale locks (see the section called “Sometimes You Just Need to Clean Up”) found in the working copy.

For each updated item, it prints a line that starts with a character reporting the action taken. These characters have the following meaning:

A
: Added

D
: Deleted

U
: Updated

C
: Conflicted

G
: Merged

A character in the first column signifies an update to the actual file, while updates to the file's properties are shown in the second column.

Alternate Names

up

Changes

Working copy

Accesses Repository

Yes

Options

```
--revision (-r) REV
--non-recursive (-N)
```

```
--quiet (-q)
--no-ignore
--incremental
--diff3-cmd CMD
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
--ignore-externals
```

Examples

Pick up repository changes that have happened since your last update:

```
$ svn update
A  newdir/toggle.c
A  newdir/disclose.c
A  newdir/launch.c
D  newdir/README
Updated to revision 32.
```

You can also “update” your working copy to an older revision (Subversion doesn't have the concept of “sticky” files like CVS does; see Appendix B, *Subversion for CVS Users*):

```
$ svn update -r30
A  newdir/README
D  newdir/toggle.c
D  newdir/disclose.c
D  newdir/launch.c
U  foo.c
Updated to revision 30.
```



If you want to examine an older revision of a single file, you may want to use **svn cat** instead—it won't change your working copy.

svnadmin

svnadmin is the administrative tool for monitoring and repairing your Subversion repository. For detailed information, see the section called “svnadmin”.

Since **svnadmin** works via direct repository access (and thus can only be used on the machine that holds the repository), it refers to the repository with a path, not a URL.

svnadmin Options

`--bdb-log-keep`
: (Berkeley DB specific) Disable automatic log removal of database log files. Having these log files around can be convenient if you need to restore from a catastrophic repository failure.

`--bdb-txn-nosync`
(Berkeley DB specific) Disables fsync when committing database transactions. Used with the **svnadmin create** command to create a Berkeley DB backed repository with `DB_TXN_NOSYNC` enabled (which improves speed but has some risks associated with it).

`--bypass-hooks`
Bypass the repository hook system.

`--clean-logs`
Removes unused Berkeley DB logs.

`--force-uuid`
By default, when loading data into repository that already contains revisions, **svnadmin** will ignore the `UUID` from the dump stream. This option will cause the repository's `UUID` to be set to the `UUID` from the stream.

`--ignore-uuid`
By default, when loading an empty repository, **svnadmin** will ignore the `UUID` from the dump stream. This option will force that UUID to be ignored (useful for overriding your configuration file if it has `--force-uuid` set).

`--incremental`
Dump a revision only as a diff against the previous revision, instead of the usual fulltext.

`--parent-dir` *`DIR`*
When loading a dump file, root paths at *`DIR`* instead of `/`.

`--revision` (`-r`) *`ARG`*
Specify a particular revision to operate on.

`--quiet`
Do not show normal progress—show only errors.

`--use-post-commit-hook`
When loading a dump file, run the repository's post-commit hook after finalizing each newly loaded revision.

`--use-pre-commit-hook`
When loading a dump file, run the repository's pre-commit hook before finalizing each newly loaded revision. If the hook fails, abort the commit and terminate the load process.

svnadmin Subcommands

Name

svnadmin create — Create a new, empty repository.

Synopsis

```
svnadmin create REPOS_PATH
```

Description

Create a new, empty repository at the path provided. If the provided directory does not exist, it will be created for you.[1] As of Subversion 1.2, **svnadmin** creates new repositories with the `fsfs` filesystem backend by default.

Options

```
--bdb-txn-nosync
--bdb-log-keep
--config-dir DIR
--fs-type TYPE
```

Examples

Creating a new repository is just this easy:

```
$ svnadmin create /usr/local/svn/repos
```

In Subversion 1.0, a Berkeley DB repository is always created. In Subversion 1.1, a Berkeley DB repository is the default repository type, but an FSFS repository can be created using the `--fs-type` option:

```
$ svnadmin create /usr/local/svn/repos --fs-type fsfs
```

[1] Remember, **svnadmin** works only with local *paths*, not *URLs*.

Name

svnadmin deltify — Deltify changed paths in a revision range.

Synopsis

```
svnadmin deltify [-r LOWER[:UPPER]] REPOS_PATH
```

Description

svnadmin deltify exists in current versions of Subversion only for historical reasons. This command is deprecated and no longer needed.

It dates from a time when Subversion offered administrators greater control over compression strategies in the repository. This turned out to be a lot of complexity for *very* little gain, and this “feature” was deprecated.

Options

```
--revision (-r) REV
--quiet (-q)
```

Name

svnadmin dump — Dump the contents of filesystem to stdout.

Synopsis

```
svnadmin dump REPOS_PATH [-r LOWER[:UPPER]] [--incremental]
```

Description

Dump the contents of filesystem to stdout in a “dumpfile” portable format, sending feedback to stderr. Dump revisions *LOWER* rev through *UPPER* rev. If no revisions are given, dump all revision trees. If only *LOWER* is given, dump that one revision tree. See the section called “Migrating Repository Data Elsewhere” for a practical use.

By default, the Subversion dumpfile stream contains a single revision (the first revision in the requested revision range) in which every file and directory in the repository in that revision is presented as if that whole tree was added at once, followed by other revisions (the remainder of the revisions in the requested range) which contain only the files and directories which were modified in those revisions. For a modified file, the complete fulltext representation of its contents, as well as all of its properties, are presented in the dumpfile; for a directory, all of its properties are presented.

There are two useful options which modify the dumpfile generator's behavior. The first is the `--incremental` option, which simply causes that first revision in the dumpfile stream to contain only the files and directories modified in that revision, instead of being presented as the addition of a new tree, and in exactly the same way that every other revision in the dumpfile is presented. This is useful for generating a relatively small dumpfile to be loaded into another repository which already has the files and directories that exist in the original repository.

The second useful option is `--deltas`. This option causes **svnadmin dump** to, instead of emitting fulltext representations of file contents and property lists, emit only deltas of those items against their previous versions. This reduces (in some cases, drastically) the size of the dumpfile that **svnadmin dump** creates. There are, however, disadvantages to using this option—deltified dumpfiles are more CPU intensive to create, cannot be operated on by **svndumpfilter**, and tend not to compress as well as their non-deltified counterparts when using third-party tools like **gzip** and **bzip2**.

Options

```
--revision (-r) REV
--incremental
--quiet (-q)
--deltas
```

Examples

Dump your whole repository:

```
$ svnadmin dump /usr/local/svn/repos
SVN-fs-dump-format-version: 1
Revision-number: 0
* Dumped revision 0.
Prop-content-length: 56
```

```
Content-length: 56
…
```

Incrementally dump a single transaction from your repository:

```
$ svnadmin dump /usr/local/svn/repos -r 21 --incremental
* Dumped revision 21.
SVN-fs-dump-format-version: 1
Revision-number: 21
Prop-content-length: 101
Content-length: 101
…
```

Name

svnadmin help — Help!

Synopsis

```
svnadmin help [SUBCOMMAND...]
```

Description

This subcommand is useful when you're trapped on a desert island with neither a net connection nor a copy of this book.

Alternate Names

?, h

Name

svnadmin hotcopy — Make a hot copy of a repository.

Synopsis

```
svnadmin hotcopy REPOS_PATH NEW_REPOS_PATH
```

Description

This subcommand makes a full “hot” backup of your repository, including all hooks, configuration files, and, of course, database files. If you pass the `--clean-logs` option, **svnadmin** will perform a hotcopy of your repository, and then remove unused Berkeley DB logs from the original repository. You can run this command at any time and make a safe copy of the repository, regardless of whether other processes are using the repository.

Options

`--clean-logs`



As described in the section called “Berkeley DB”, hot-copied Berkeley DB repositories are *not* portable across operating systems, nor will they work on machines with a different “endianness” than the machine where they were created.

Name

svnadmin list-dblogs — Ask Berkeley DB which log files exist for a given Subversion repository (applies only to repositories using the bdb backend).

Synopsis

```
svnadmin list-dblogs REPOS_PATH
```

Description

Berkeley DB creates logs of all changes to the repository, which allow it to recover in the face of catastrophe. Unless you enable DB_LOG_AUTOREMOVE, the log files accumulate, although most are no longer used and can be deleted to reclaim disk space. See the section called “Managing Disk Space” for more information.

Name

svnadmin list-unused-dblogs — Ask Berkeley DB which log files can be safely deleted (applies only to repositories using the `bdb` backend).

Synopsis

```
svnadmin list-unused-dblogs REPOS_PATH
```

Description

Berkeley DB creates logs of all changes to the repository, which allow it to recover in the face of catastrophe. Unless you enable `DB_LOG_AUTOREMOVE`, the log files accumulate, although most are no longer used and can be deleted to reclaim disk space. See the section called “Managing Disk Space” for more information.

Examples

Remove all unused log files from a repository:

```
$ svnadmin list-unused-dblogs /path/to/repos
/path/to/repos/log.0000000031
/path/to/repos/log.0000000032
/path/to/repos/log.0000000033

$ svnadmin list-unused-dblogs /path/to/repos | xargs rm
## disk space reclaimed!
```

Name

svnadmin load — Read a “dumpfile”-formatted stream from stdin.

Synopsis

```
svnadmin load REPOS_PATH
```

Description

Read a “dumpfile”-formatted stream from stdin, committing new revisions into the repository's filesystem. Send progress feedback to stdout.

Options

```
--quiet (-q)
--ignore-uuid
--force-uuid
--use-pre-commit-hook
--use-post-commit-hook
--parent-dir
```

Example

This shows the beginning of loading a repository from a backup file (made, of course, with **svnadmin dump**):

```
$ svnadmin load /usr/local/svn/restored < repos-backup
<<< Started new txn, based on original revision 1
     * adding path : test ... done.
     * adding path : test/a ... done.
…
```

Or if you want to load into a subdirectory:

```
$ svnadmin load --parent-dir new/subdir/for/project /usr/local/svn/restored < repo
<<< Started new txn, based on original revision 1
     * adding path : test ... done.
     * adding path : test/a ... done.
…
```

Name

svnadmin lslocks — Print descriptions of all locks.

Synopsis

```
svnadmin lslocks REPOS_PATH
```

Description

Print descriptions of all locks in a repository.

Options

None

Example

This lists the one locked file in the repository at `/svn/repos`:

```
$ svnadmin lslocks /svn/repos
Path: /tree.jpg
UUID Token: opaquelocktoken:ab00ddf0-6afb-0310-9cd0-dda813329753
Owner: harry
Created: 2005-07-08 17:27:36 -0500 (Fri, 08 Jul 2005)
Expires:
Comment (1 line):
Rework the uppermost branches on the bald cypress in the foreground.
```

Name

svnadmin lstxns — Print the names of all uncommitted transactions.

Synopsis

```
svnadmin lstxns REPOS_PATH
```

Description

Print the names of all uncommitted transactions. See the section called “Removing dead transactions” for information on how uncommitted transactions are created and what you should do with them.

Examples

List all outstanding transactions in a repository.

```
$ svnadmin lstxns /usr/local/svn/repos/
1w
1x
```

Name

svnadmin recover — Bring a repository database back into a consistent state (applies only to repositories using the `bdb` backend). In addition, if `repos/conf/passwd` does not exist, it will create a default password file .

Synopsis

```
svnadmin recover REPOS_PATH
```

Description

Run this command if you get an error indicating that your repository needs to be recovered.

Options

```
--wait
```

Examples

Recover a hung repository:

```
$ svnadmin recover /usr/local/svn/repos/
Repository lock acquired.
Please wait; recovering the repository may take some time...

Recovery completed.
The latest repos revision is 34.
```

Recovering the database requires an exclusive lock on the repository. (This is a “database lock”; see The three meanings of “lock”.) If another process is accessing the repository, then **svnadmin recover** will error:

```
$ svnadmin recover /usr/local/svn/repos
svn: Failed to get exclusive repository access; perhaps another process
such as httpd, svnserve or svn has it open?

$
```

The `--wait` option, however, will cause **svnadmin recover** to wait indefinitely for other processes to disconnect:

```
$ svnadmin recover /usr/local/svn/repos --wait
Waiting on repository lock; perhaps another process has it open?

### time goes by…

Repository lock acquired.
Please wait; recovering the repository may take some time...

Recovery completed.
The latest repos revision is 34.
```

Name

svnadmin rmlocks — Unconditionally remove one or more locks from a repository.

Synopsis

```
svnadmin rmlocks REPOS_PATH LOCKED_PATH...
```

Description

Remove lock from each *LOCKED_PATH*.

Options

None

Example

This deletes the locks on `tree.jpg` and `house.jpg` in the repository at `/svn/repos`

```
$ svnadmin rmlocks /svn/repos tree.jpg house.jpg
Removed lock on '/tree.jpg.
Removed lock on '/house.jpg.
```

Name

svnadmin rmtxns — Delete transactions from a repository.

Synopsis

```
svnadmin rmtxns REPOS_PATH TXN_NAME...
```

Description

Delete outstanding transactions from a repository. This is covered in detail in the section called “Removing dead transactions”.

Options

```
--quiet (-q)
```

Examples

Remove named transactions:

```
$ svnadmin rmtxns /usr/local/svn/repos/ 1w 1x
```

Fortunately, the output of **lstxns** works great as the input for **rmtxns**:

```
$ svnadmin rmtxns /usr/local/svn/repos/  `svnadmin lstxns /usr/local/svn/repos/`
```

Which will remove all uncommitted transactions from your repository.

Name

svnadmin setlog — Set the log-message on a revision.

Synopsis

```
svnadmin setlog REPOS_PATH -r REVISION FILE
```

Description

Set the log-message on revision REVISION to the contents of FILE.

This is similar to using **svn propset --revprop** to set the `svn:log` property on a revision, except that you can also use the option `--bypass-hooks` to avoid running any pre- or post-commit hooks, which is useful if the modification of revision properties has not been enabled in the pre-revprop-change hook.



Revision properties are not under version control, so this command will permanently overwrite the previous log message.

Options

```
--revision (-r) REV
--bypass-hooks
```

Examples

Set the log message for revision 19 to the contents of the file `msg`:

```
$ svnadmin setlog /usr/local/svn/repos/ -r 19 msg
```

Name

svnadmin verify — Verify the data stored in the repository.

Synopsis

```
svnadmin verify REPOS_PATH
```

Description

Run this command if you wish to verify the integrity of your repository. This basically iterates through all revisions in the repository by internally dumping all revisions and discarding the output—it's a good idea to run this on a regular basis to guard against latent hard disk failures and “bitrot”. If this command fails—which it will do at the first sign of a problem—that means that your repository has at least one corrupted revision and you should restore the corrupted revision from a backup (you did make a backup, didn't you?).

Examples

Verify a hung repository:

```
$ svnadmin verify /usr/local/svn/repos/
* Verified revision 1729.
```

svnlook

svnlook is a command-line utility for examining different aspects of a Subversion repository. It does not make any changes to the repository—it's just used for “peeking”. **svnlook** is typically used by the repository hooks, but a repository administrator might find it useful for diagnostic purposes.

Since **svnlook** works via direct repository access (and thus can only be used on the machine that holds the repository), it refers to the repository with a path, not a URL.

If no revision or transaction is specified, **svnlook** defaults to the youngest (most recent) revision of the repository.

svnlook Options

Options in **svnlook** are global, just like in **svn** and **svnadmin**; however, most options only apply to one subcommand since the functionality of **svnlook** is (intentionally) limited in scope.

`--no-diff-deleted`
: Prevents **svnlook** from printing differences for deleted files. The default behavior when a file is deleted in a transaction/revision is to print the same differences that you would see if you had left the file but removed all the content.

`--revision` (`-r`)
: Specify a particular revision number that you wish to examine.

`--revprop`
: Operates on a revision property instead of a property specific to a file or directory. This op-

tion requires that you also pass a revision with the `--revision` (`-r`) option.

`--transaction` (`-t`)
Specify a particular transaction ID that you wish to examine.

`--show-ids`
Show the filesystem node revision IDs for each path in the filesystem tree.

svnlook Subcommands

Name

svnlook author — Print the author.

Synopsis

```
svnlook author REPOS_PATH
```

Description

Print the author of a revision or transaction in the repository.

Options

```
--revision (-r) REV
--transaction (-t)
```

Examples

svnlook author is handy, but not very exciting:

```
$ svnlook author -r 40 /usr/local/svn/repos
sally
```

Name

svnlook cat — Print the contents of a file.

Synopsis

```
svnlook cat REPOS_PATH PATH_IN_REPOS
```

Description

Print the contents of a file.

Options

```
--revision (-r) REV
--transaction (-t)
```

Examples

This shows the contents of a file in transaction `ax8`, located at `/trunk/README`:

```
$ svnlook cat -t ax8 /usr/local/svn/repos /trunk/README

               Subversion, a version control system.
               =====================================

$LastChangedDate: 2003-07-17 10:45:25 -0500 (Thu, 17 Jul 2003) $

Contents:

     I. A FEW POINTERS
    II. DOCUMENTATION
   III. PARTICIPATING IN THE SUBVERSION COMMUNITY
…
```

Name

svnlook changed — Print the paths that were changed.

Synopsis

```
svnlook changed REPOS_PATH
```

Description

Print the paths that were changed in a particular revision or transaction, as well as “svn update-style” status letters in the first two columns:

'A '
: Item added to repository.

'D '
: Item deleted from repository.

'U '
: File contents changed.

' U'
: Properties of item changed. Note the leading space.

'UU'
: File contents and properties changed.

Files and directories can be distinguished, as directory paths are displayed with a trailing '/' character.

Options

```
--revision (-r) REV
--transaction (-t)
```

Examples

This shows a list of all the changed files and directories in revision 39 of a test repository. Note that the first changed item is a directory, as evidenced by the trailing /:

```
$ svnlook changed -r 39 /usr/local/svn/repos
A   trunk/vendors/deli/
A   trunk/vendors/deli/chips.txt
A   trunk/vendors/deli/sandwich.txt
A   trunk/vendors/deli/pickle.txt
U   trunk/vendors/baker/bagel.txt
 U  trunk/vendors/baker/croissant.txt
UU  trunk/vendors/baker/pretzel.txt
D   trunk/vendors/baker/baguette.txt
```

Name

svnlook date — Print the datestamp.

Synopsis

```
svnlook date REPOS_PATH
```

Description

Print the datestamp of a revision or transaction in a repository.

Options

```
--revision (-r) REV
--transaction (-t)
```

Examples

This shows the date of revision 40 of a test repository:

```
$ svnlook date -r 40 /tmp/repos/
2003-02-22 17:44:49 -0600 (Sat, 22 Feb 2003)
```

Name

svnlook diff — Print differences of changed files and properties.

Synopsis

```
svnlook diff REPOS_PATH
```

Description

Print GNU-style differences of changed files and properties in a repository.

Options

```
--revision (-r) REV
--transaction (-t)
--no-diff-added
--no-diff-deleted
```

Examples

This shows a newly added (empty) file, a deleted file, and a copied file:

```
$ svnlook diff -r 40 /usr/local/svn/repos/
Copied: egg.txt (from rev 39, trunk/vendors/deli/pickle.txt)

Added: trunk/vendors/deli/soda.txt
===================================================================================

Modified: trunk/vendors/deli/sandwich.txt
===================================================================================
--- trunk/vendors/deli/sandwich.txt (original)
+++ trunk/vendors/deli/sandwich.txt 2003-02-22 17:45:04.000000000 -0600
@@ -0,0 +1 @@
+Don't forget the mayo!

Modified: trunk/vendors/deli/logo.jpg
===================================================================================
(Binary files differ)

Deleted: trunk/vendors/deli/chips.txt
===================================================================================

Deleted: trunk/vendors/deli/pickle.txt
===================================================================================
```

If a file has a non-textual `svn:mime-type` property, then the differences are not explicitly shown.

Name

svnlook dirs-changed — Print the directories that were themselves changed.

Synopsis

```
svnlook dirs-changed REPOS_PATH
```

Description

Print the directories that were themselves changed (property edits) or whose file children were changed.

Options

```
--revision (-r) REV
--transaction (-t)
```

Examples

This shows the directories that changed in revision 40 in our sample repository:

```
$ svnlook dirs-changed -r 40 /usr/local/svn/repos
trunk/vendors/deli/
```

Name

svnlook help — Help!

Synopsis

```
Also svnlook -h and svnlook -?.
```

Description

Displays the help message for svnlook. This command, like its brother **svn help**, is also your friend, even though you never call it anymore and forgot to invite it to your last party.

Alternate Names

?, h

Name

svnlook history — Print information about the history of a path in the repository (or the root directory if no path is supplied).

Synopsis

```
svnlook history REPOS_PATH [PATH_IN_REPOS]
```

Description

Print information about the history of a path in the repository (or the root directory if no path is supplied).

Options

```
--revision (-r) REV
--show-ids
```

Examples

This shows the history output for the path `/tags/1.0` as of revision 20 in our sample repository.

```
$ svnlook history -r 20 /usr/local/svn/repos /tags/1.0 --show-ids
REVISION   PATH <ID>
--------   ---------
      19   /tags/1.0 <1.2.12>
      17   /branches/1.0-rc2 <1.1.10>
      16   /branches/1.0-rc2 <1.1.x>
      14   /trunk <1.0.q>
      13   /trunk <1.0.o>
      11   /trunk <1.0.k>
       9   /trunk <1.0.g>
       8   /trunk <1.0.e>
       7   /trunk <1.0.b>
       6   /trunk <1.0.9>
       5   /trunk <1.0.7>
       4   /trunk <1.0.6>
       2   /trunk <1.0.3>
       1   /trunk <1.0.2>
```

Name

svnlook info — Print the author, datestamp, log message size, and log message.

Synopsis

```
svnlook info REPOS_PATH
```

Description

Print the author, datestamp, log message size, and log message.

Options

```
--revision (-r) REV
--transaction (-t)
```

Examples

This shows the info output for revision 40 in our sample repository.

```
$ svnlook info -r 40 /usr/local/svn/repos
sally
2003-02-22 17:44:49 -0600 (Sat, 22 Feb 2003)
15
Rearrange lunch.
```

Name

svnlook lock — If a lock exists on a path in the repository, describe it.

Synopsis

```
svnlook lock REPOS_PATH PATH_IN_REPOS
```

Description

Print all information available for the lock at *`PATH_IN_REPOS`*. If *`PATH_IN_REPOS`* is not locked, print nothing.

Options

None

Examples

This describes the lock on the file `tree.jpg`.

```
$ svnlook lock /svn/repos tree.jpg
UUID Token: opaquelocktoken:ab00ddf0-6afb-0310-9cd0-dda813329753
Owner: harry
Created: 2005-07-08 17:27:36 -0500 (Fri, 08 Jul 2005)
Expires:
Comment (1 line):
Rework the uppermost branches on the bald cypress in the foreground.
```

Name

svnlook log — Print the log message.

Synopsis

```
svnlook log REPOS_PATH
```

Description

Print the log message.

Options

```
--revision (-r) REV
--transaction (-t)
```

Examples

This shows the log output for revision 40 in our sample repository:

```
$ svnlook log /tmp/repos/
Rearrange lunch.
```

Name

svnlook propget — Print the raw value of a property on a path in the repository.

Synopsis

```
svnlook propget REPOS_PATH PROPNAME [PATH_IN_REPOS]
```

Description

List the value of a property on a path in the repository.

Alternate Names

pg, pget

Options

```
--revision (-r) REV
--transaction (-t)
--revprop
```

Examples

This shows the value of the “seasonings” property on the file `/trunk/sandwich` in the `HEAD` revision:

```
$ svnlook pg /usr/local/svn/repos seasonings /trunk/sandwich
mustard
```

Name

svnlook proplist — Print the names and values of versioned file and directory properties.

Synopsis

```
svnlook proplist REPOS_PATH [PATH_IN_REPOS]
```

Description

List the properties of a path in the repository. With `--verbose`, show the property values too.

Alternate Names

pl, plist

Options

```
--revision (-r) REV
--transaction (-t)
--verbose (-v)
--revprop
```

Examples

This shows the names of properties set on the file `/trunk/README` in the `HEAD` revision:

```
$ svnlook proplist /usr/local/svn/repos /trunk/README
  original-author
  svn:mime-type
```

This is the same command as in the previous example, but this time showing the property values as well:

```
$ svnlook --verbose proplist /usr/local/svn/repos /trunk/README
  original-author : fitz
  svn:mime-type : text/plain
```

Name

svnlook tree — Print the tree.

Synopsis

```
svnlook tree REPOS_PATH [PATH_IN_REPOS]
```

Description

Print the tree, starting at *PATH_IN_REPOS* (if supplied, at the root of the tree otherwise), optionally showing node revision IDs.

Options

```
--revision (-r) REV
--transaction (-t)
--show-ids
```

Examples

This shows the tree output (with node-IDs) for revision 40 in our sample repository:

```
$ svnlook tree -r 40 /usr/local/svn/repos --show-ids
/ <0.0.2j>
 trunk/ <p.0.2j>
  vendors/ <q.0.2j>
   deli/ <1g.0.2j>
    egg.txt <1i.e.2j>
    soda.txt <1k.0.2j>
    sandwich.txt <1j.0.2j>
```

Name

svnlook uuid — Print the repository's `UUID`.

Synopsis

```
svnlook uuid REPOS_PATH
```

Description

Print the `UUID` for the repository. the UUID is the repository's *u*niversal *u*nique *id*entifier. The Subversion client uses this identifier to differentiate between one repository and another.

Examples

```
$ svnlook uuid /usr/local/svn/repos
e7fe1b91-8cd5-0310-98dd-2f12e793c5e8
```

Name

svnlook youngest — Print the youngest revision number.

Synopsis

```
svnlook youngest REPOS_PATH
```

Description

Print the youngest revision number of a repository.

Examples

This shows the youngest revision of our sample repository:

```
$ svnlook youngest /tmp/repos/
42
```

svnsync

svnsync is the Subversion remote repository mirroring tool. Put simply, it allows you to replay the revisions of one repository into another one.

In any mirroring scenario, there are two repositories: the source repository, and the mirror (or “sink”) repository. The source repository is the repository from which **svnsync** pulls revisions. The mirror repository is the destination for the revisions pulled from the source repository. Each of the repositories may be local or remote—they are only ever addressed by their URLs.

The **svnsync** process requires only read access to the source repository; it never attempts to modify it. But obviously, **svnsync** requires both read and write access to the mirror repository.



svnsync is very sensitive to changes made in the mirror repository that weren't made as part of a mirroring operation. To prevent this from happening, it's best if the **svnsync** process is the only process permitted to modify the mirror repository.

svnsync Options

`--config-dir` *`DIR`*
: Instructs Subversion to read configuration information from the specified directory instead of the default location (`.subversion` in the user's home directory).

`--no-auth-cache`
: Prevents caching of authentication information (e.g. username and password) in the Subversion administrative directories.

`--non-interactive`
: In the case of an authentication failure, or insufficient credentials, prevents prompting for credentials (e.g. username or password). This is useful if you're running Subversion inside of an automated script and it's more appropriate to have Subversion fail than to prompt for

more information.

`--password` *PASS*
: Indicates that you are providing your password for authentication on the command line—otherwise, if it is needed, Subversion will prompt you for it.

`--username` *NAME*
: Indicates that you are providing your username for authentication on the command line—otherwise, if it is needed, Subversion will prompt you for it.

svnsync Subcommands

Here are the various subcommands:

Name

svnsync copy-revprops — Copy all revision properties for a given revision from the source repository to the mirror repository.

Synopsis

```
svnsync copy-revprops DEST_URL REV
```

Description

Because Subversion revision properties can be changed at any time, it's possible that the properties for some revision might be changed after that revision has already been synchronized to another repository. Because the **svnsync synchronize** command operates only on the range of revisions that have not yet been synchronized, it won't notice a revision property change outside that range. Left as is, this causes a deviation in the values of that revision's properties between the source and mirror repositories. **svnsync copy-revprops** is the answer to this problem. Use it to re-synchronize the revision properties for a particular revision.

Alternate Names

None

Options

```
--non-interactive
--no-auth-cache
--username NAME
--password PASS
--config-dir DIR
```

Examples

Re-synchronize revision properties for a single revision:

```
$ svnsync copy-revprops file:///opt/svn/repos-mirror 6
Copied properties for revision 6.
$
```

Name

svnsync initialize — Initialize a destination repository for synchronization from another repository.

Synopsis

```
svnsync initialize DEST_URL SOURCE_URL
```

Description

svnsync initialize verifies that a repository meets the requirements of a new mirror repository—that it has no previous existing version history, and that it allows revision property modifications—and records the initial administrative information which associates the mirror repository with the source repository. This is the first **svnsync** operation you run on a would-be mirror repository.

Alternate Names

init

Options

```
--non-interactive
--no-auth-cache
--username NAME
--password PASS
--config-dir DIR
```

Examples

Fail to initialize a mirror repository due to inability to modify revision properties:

```
$ svnsync initialize file:///opt/svn/repos-mirror http://svn.example.com/repos
svnsync: Repository has not been enabled to accept revision propchanges;
ask the administrator to create a pre-revprop-change hook
$
```

Initialize a repository as a mirror, having already created a pre-revprop-change hook which permits all revision property changes:

```
$ svnsync initialize file:///opt/svn/repos-mirror http://svn.example.com/repos
Copied properties for revision 0.
$
```

Name

svnsync synchronize — Transfer all pending revisions from the source repository to the mirror repository.

Synopsis

```
svnsync synchronize DEST_URL
```

Description

The **svnsync synchronize** command does all the heavy lifting of a repository mirroring operation. After consulting with the mirror repository to see which revisions have already been copied into it, it then begins copying any not-yet-mirrored revisions from the source repository.

svnsync synchronize can be gracefully cancelled and restarted.

Alternate Names

sync

Options

```
--non-interactive
--no-auth-cache
--username NAME
--password PASS
--config-dir DIR
```

Examples

Copy unsynchronized revisions from the source repository to the mirror repository:

```
$ svnsync synchronize file:///opt/svn/repos-mirror
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Committed revision 3.
Copied properties for revision 3.
…
Committed revision 45.
Copied properties for revision 45.
Committed revision 46.
Copied properties for revision 46.
Committed revision 47.
Copied properties for revision 47.
$
```

svnserve

svnserve allows access to Subversion repositories using Subversion's custom network protocol.

You can run **svnserve** as a standalone server process (for clients that are using the `svn://` access method); you can have a daemon such as **inetd** or **xinetd** launch it for you on demand (also for `svn://`), or you can have **sshd** launch it on demand for the `svn+ssh://` access method.

Regardless of the access method, once the client has selected a repository by transmitting its URL, **svnserve** reads a file named `conf/svnserve.conf` in the repository directory to determine repository-specific settings such as what authentication database to use and what authorization policies to apply. See the section called “svnserve, a custom server” for details of the `svnserve.conf` file.

svnserve Options

Unlike the previous commands we've described, **svnserve** has no subcommands—**svnserve** is controlled exclusively by options.

`--daemon` (`-d`)
: Causes **svnserve** to run in daemon mode. **svnserve** backgrounds itself and accepts and serves TCP/IP connections on the svn port (3690, by default).

`--listen-port=`*`PORT`*
: Causes svnserve to listen on *`PORT`* when run in daemon mode. (FreeBSD daemons only listen on tcp6 by default—this option tells them to also listen on tcp4.)

`--listen-host=`*`HOST`*
: Causes **svnserve** to listen on the interface specified by *`HOST`*, which may be either a hostname or an IP address.

`--foreground`
: When used together with `-d`, this option causes **svnserve** to stay in the foreground. This option is mainly useful for debugging.

`--inetd` (`-i`)
: Causes **svnserve** to use the stdin/stdout file descriptors, as is appropriate for a daemon running out of **inetd**.

`--help` (`-h`)
: Displays a usage summary and exits.

`--version`
: Displays version information, a list of repository back-end modules available, and exits.

`--root=`*`ROOT`* (`-r=`*`ROOT`*)
: Sets the virtual root for repositories served by **svnserve**. The pathname in URLs provided by the client will be interpreted relative to this root, and will not be allowed to escape this root.

`--tunnel` (`-t`)
: Causes **svnserve** to run in tunnel mode, which is just like the **inetd** mode of operation (both modes serve one connection over stdin/stdout, then exit), except that the connection is considered to be pre-authenticated with the username of the current uid. This flag is automatically passed for you by the client when running over a tunnel agent such as **ssh**. That means there's rarely any need for *you* to pass this option to **svnserve**. So if you find yourself typing `svnserve --tunnel` on the command line, and wondering what to do next, see the section called “Tunneling over SSH”.

`--tunnel-user NAME`
Used in conjunction with the `--tunnel` option; tells svnserve to assume that *NAME* is the authenticated user, rather than the UID of the svnserve process. Useful for users wishing to share a single system account over SSH, but maintaining separate commit identities.

`--threads` (`-T`)
When running in daemon mode, causes **svnserve** to spawn a thread instead of a process for each connection (e.g. for when running on Windows). The **svnserve** process still backgrounds itself at startup time.

`--listen-once` (`-X`)
Causes **svnserve** to accept one connection on the svn port, serve it, and exit. This option is mainly useful for debugging.

svnversion

Name

svnversion — Summarize the local revision(s) of a working copy.

Synopsis

```
svnversion [OPTIONS] [WC_PATH [TRAIL_URL]]
```

Description

svnversion is a program for summarizing the revision mixture of a working copy. The resultant revision number, or revision range, is written to standard output.

It's common to use this output in your build process when defining the version number of your program.

`TRAIL_URL`, if present, is the trailing portion of the URL used to determine if *`WC_PATH`* itself is switched (detection of switches within *`WC_PATH`* does not rely on *`TRAIL_URL`*).

When *`WC_PATH`* is not defined, the current directory will be used as the working copy path. *`TRAIL_URL`* cannot be defined if *`WC_PATH`* is not explicitly given.

Options

Like **svnserve**, **svnversion** has no subcommands, it only has options.

`--no-newline` (`-n`)
: Omit the usual trailing newline from the output.

`--committed` (`-c`)
: Use the last-changed revisions rather than the current (i.e., highest locally available) revisions.

`--help` (`-h`)
: Print a help summary.

`--version`
: Print the version of **svnversion** and exit with no error.

Examples

If the working copy is all at the same revision (for example, immediately after an update), then that revision is printed out:

```
$ svnversion
4168
```

You can add *`TRAIL_URL`* to make sure that the working copy is not switched from what you expect. Note that the *`WC_PATH`* is required in this command:

```
$ svnversion . /repos/svn/trunk
```

```
4168
```

For a mixed-revision working copy, the range of revisions present is printed:

```
$ svnversion
4123:4168
```

If the working copy contains modifications, a trailing "M" is added:

```
$ svnversion
4168M
```

If the working copy is switched, a trailing "S" is added:

```
$ svnversion
4168S
```

Thus, here is a mixed-revision, switched working copy containing some local modifications:

```
$ svnversion
4212:4168MS
```

If invoked on a directory that is not a working copy, **svnversion** assumes it is an exported working copy and prints "exported":

```
$ svnversion
exported
```

mod_dav_svn

Name

mod_dav_svn Configuration Directives — Apache configuration directives for serving Subversion repositories through Apache HTTP Server.

Description

This section briefly describes each of the Subversion Apache configuration directives. For an in-depth description of configuring Apache with Subversion, see the section called “httpd, the Apache HTTP server”.)

Directives

`DAV svn`
: This directive must be included in any `Directory` or `Location` block for a Subversion repository. It tells httpd to use the Subversion backend for mod_dav to handle all requests.

`SVNAutoversioning On`
: This directive allows write requests from WebDAV clients to result in automatic commits. A generic log message is auto-generated and attached to each revision. If you enable Autoversioning, you'll likely want to set `ModMimeUsePathInfo On` so that `mod_mime` can set `svn:mime-type` to the correct mime-type automatically (as best as `mod_mime` is able to, of course). For more information, see Appendix C, *WebDAV and Autoversioning*

`SVNPath`
: This directive specifies the location in the filesystem for a Subversion repository's files. In a configuration block for a Subversion repository, either this directive or `SVNParentPath` must be present, but not both.

`SVNSpecialURI`
: Specifies the URI component (namespace) for special Subversion resources. The default is “`!svn`”, and most administrators will never use this directive. Only set this if there is a pressing need to have a file named `!svn` in your repository. If you change this on a server already in use, it will break all of the outstanding working copies and your users will hunt you down with pitchforks and flaming torches.

`SVNReposName`
: Specifies the name of a Subversion repository for use in `HTTP GET` responses. This value will be prepended to the title of all directory listings (which are served when you navigate to a Subversion repository with a web browser). This directive is optional.

`SVNIndexXSLT`
: Specifies the URI of an XSL transformation for directory indexes. This directive is optional.

`SVNParentPath`
: Specifies the location in the filesystem of a parent directory whose child directories are Subversion repositories. In a configuration block for a Subversion repository, either this directive or `SVNPath` must be present, but not both.

`SVNPathAuthz`
: Control path-based authorization by enabling or disabling subrequests. See the section called “Disabling Path-based Checks” for details.

Subversion properties

Subversion allows users to invent arbitrarily-named versioned properties on files and directories, as well as unversioned properties on revisions. The only restriction is on properties whose names begin with `svn:` (those are reserved for Subversion's own use). While these properties may be set by users to control Subversion's behavior, users may not invent new `svn:` properties.

Versioned Properties

`svn:executable`
: If present on a file, the client will make the file executable in Unix-hosted working copies. See the section called “File Executability”.

`svn:mime-type`
: If present on a file, the value indicates the file's mime-type. This allows the client to decide whether line-based contextual merging is safe to perform during updates, and can also affect how the file behaves when fetched via web browser. See the section called “File Content Type”.

`svn:ignore`
: If present on a directory, the value is a list of *unversioned* file patterns to be ignored by **svn status** and other subcommands. See the section called “Ignoring Unversioned Items”

`svn:keywords`
: If present on a file, the value tells the client how to expand particular keywords within the file. See the section called “Keyword Substitution”.

`svn:eol-style`
: If present on a file, the value tells the client how to manipulate the file's line-endings in the working copy, and in exported trees. See the section called “End-of-Line Character Sequences” and svn export.

`svn:externals`
: If present on a directory, the value is a multi-line list of other paths and URLs the client should check out. See the section called “Externals Definitions”.

`svn:special`
: If present on a file, indicates that the file is not an ordinary file, but a symbolic link or other special object[1].

`svn:needs-lock`
: If present on a file, tells the client to make the file read-only in the working copy, as a reminder that the file should be locked before editing begins. See the section called “Lock Communication”.

Unversioned Properties

`svn:author`
: If present, contains the authenticated username of the person who created the revision. (If not present, then the revision was committed anonymously.)

[1] As of this writing, symbolic links are indeed the only “special” objects. But there might be more in future releases of Subversion.

`svn:date`
: Contains the UTC time the revision was created, in ISO 8601 format. The value comes from the *server* machine's clock, not the client's.

`svn:log`
: Contains the log message describing the revision.

`svn:autoversioned`
: If present, the revision was created via the autoversioning feature. See the section called “Autoversioning”.

Repository Hooks

Name

start-commit — Notification of the beginning of a commit.

Description

The start-commit hook is run before the commit transaction is even created. It is typically used to decide if the user has commit privileges at all.

If the start-commit hook program returns a non-zero exit value, the commit is stopped before the commit transaction is even created, and anything printed to stderr is marshalled back to the client.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path
2. authenticated username attempting the commit

Common Uses

access control

Name

pre-commit — Notification just prior to commit completion.

Description

The pre-commit hook is run just before a commit transaction is promoted to a new revision. Typically, this hook is used to protect against commits that are disallowed due to content or location (for example, your site might require that all commits to a certain branch include a ticket number from the bug tracker, or that the incoming log message is non-empty).

If the pre-commit hook program returns a non-zero exit value, the commit is aborted, the commit transaction is removed, and anything printed to stderr is marshalled back to the client.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path
2. commit transaction name

Common Uses

change validation and control

Name

post-commit — Notification of a successful commit.

Description

The post-commit hook is run after the transaction is committed, and a new revision created. Most people use this hook to send out descriptive emails about the commit or to notify some other tool (such as an issue tracker) that a commit has happened. Some configurations also use this hook to trigger backup processes.

The output from, and exit value returned by the post-commit hook program are ignored.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path
2. revision number created by the commit

Common Uses

commit notification, tool integration

Name

pre-revprop-change — Notification of a revision property change attempt.

Description

The pre-revprop-change hook is run immediately prior to the modification of a revision property when performed outside the scope of a normal commit. Unlike the other hooks, the default state of this one is to deny the proposed action. The hook must actually exist and return a zero exit value before a revision property modification can happen.

If the pre-revprop-change hook doesn't exist, isn't executable, or returns a non-zero exit value, no change to the property will be made, and anything printed to stderr is marshalled back to the client.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path
2. revision whose property is about to be modified
3. authenticated username attempting the propchange
4. name of the property changed
5. change description: `A` (added), `D` (deleted), or `M` (modified)

Additionally, Subversion passes to the hook program via standard input the proposed value of the property.

Common Uses

access control, change validation and control

Name

post-revprop-change — Notification of a successful revision property change.

Description

The post-revprop-change hook is run immediately after to the modification of a revision property when performed outside the scope of a normal commit. As can be derived from the description of its counterpart, the pre-revprop-change hook, this hook will not run at all unless the pre-revprop-change hook is implemented. It is typically used to send email notification of the property change.

The output from, and exit value returned by, the post-revprop-change hook program are ignored.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path
2. revision whose property was modified
3. authenticated username of the person making the change
4. name of the property changed
5. change description: `A` (added), `D` (deleted), or `M` (modified)

Additionally, Subversion passes to the hook program, via standard input, the previous value of the property.

Common Uses

propchange notification

Name

pre-lock — Notification of a path lock attempt.

Description

The pre-lock hook runs whenever someone attempts to lock a path. It can be used to prevent locks altogether, or to create a more complex policy specifying exactly which users are allowed to lock particular paths. If the hook notices a pre-existing lock, then it can also decide whether a user is allowed to “steal” the existing lock.

If the pre-lock hook program returns a non-zero exit value, the lock action is aborted and anything printed to stderr is marshalled back to the client.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path
2. versioned path which is to be locked
3. authenticated username of the person attempting the lock

Common Uses

access control

Name

post-lock — Notification of a successful path lock.

Description

The post-lock hook runs after one or more paths has been locked. It is typically used to send email notification of the lock event.

The output from and exit value returned by the post-lock hook program are ignored.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path
2. authenticated username of the person who locked the paths

Additionally, the list of paths locked is passed to the hook program via standard input, one path per line.

Common Uses

lock notification

Name

pre-unlock — Notification of a path unlock attempt.

Description

The pre-unlock hook runs whenever someone attempts to remove a lock on a file. It can be used to create policies that specify which users are allowed to unlock particular paths. It's particularly important for determining policies about lock breakage. If user A locks a file, is user B allowed to break the lock? What if the lock is more than a week old? These sorts of things can be decided and enforced by the hook.

If the pre-unlock hook program returns a non-zero exit value, the unlock action is aborted and anything printed to stderr is marshalled back to the client.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path
2. versioned path which is to be locked
3. authenticated username of the person attempting the lock

Common Uses

access control

Name

post-unlock — Notification of a successful path unlock.

Description

The post-unlock hook runs after one or more paths has been unlocked. It is typically used to send email notification of the unlock event.

The output from and exit value returned by, the post-unlock hook program are ignored.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path
2. authenticated username of the person who unlocked the paths

Additionally, the list of paths unlocked is passed to the hook program via standard input, one path per line.

Common Uses

unlock notification

Appendix A. Subversion Quick-Start Guide

If you're eager to get Subversion up and running (and you enjoy learning by experimentation), this chapter will show you how to create a repository, import code, and then check it back out again as a working copy. Along the way, we give links to the relevant chapters of this book.



If you're new to the entire concept of version control or to the “copy-modify-merge” model used by both CVS and Subversion, then you should read Chapter 1, *Fundamental Concepts* before going any further.

Installing Subversion

Subversion is built on a portability layer called APR—the Apache Portable Runtime library. The APR library provides all the interfaces that Subversion needs to function on different operating systems: disk access, network access, memory management, and so on. While Subversion is able to use Apache as one of its network server programs, its dependence on APR *does not* mean that Apache is a required component. APR is a standalone library useable by any application. It does mean, however, that like Apache, Subversion clients and servers run on any operating system that the Apache httpd server runs on: Windows, Linux, all flavors of BSD, Mac OS X, Netware, and others.

The easiest way to get Subversion is to download a binary package built for your operating system. Subversion's website (http://subversion.tigris.org) often has these packages available for download, posted by volunteers. The site usually contains graphical installer packages for users of Microsoft operating systems. If you run a Unix-like operating system, you can use your system's native package distribution system (RPMs, DEBs, the ports tree, etc.) to get Subversion.

Alternately, you can build Subversion directly from source code, though it's not always an easy task. (If you're not experienced at building open source software packages, you're probably better off downloading a binary distribution instead!) From the Subversion website, download the latest source-code release. After unpacking it, follow the instructions in the `INSTALL` file to build it. Note that a released source package may not contain everything you need to build a command-line client capable of talking to a remote repository. Starting with Subversion 1.4 and later, the libraries Subversion depends on (apr, apr-util, and neon) are distributed in a separate source package suffixed with `-deps`. These libraries are now common enough that they may already be installed on your system. If not, you'll need to unpack the dependency package into the same directory where you unpacked the main Subversion source. Regardless, it's possible that you may want to fetch other optional dependencies such as Berkeley DB and possibly Apache httpd. If you want to do a complete build, make sure you have all of the packages documented in the `INSTALL` file.

If you're one of those folks that likes to use bleeding-edge software, you can also get the Subversion source code from the Subversion repository in which it lives. Obviously, you'll need to already have a Subversion client on hand to do this. But once you do, you can check out a working copy of the Subversion source repository from http://svn.collab.net/repos/svn/trunk/: [1]

[1]Note that the URL checked out in the example above ends not with `svn`, but with a subdirectory thereof called `trunk`. See our discussion of Subversion's branching and tagging model for the reasoning behind this.

```
$ svn checkout http://svn.collab.net/repos/svn/trunk subversion
A    subversion/HACKING
A    subversion/INSTALL
A    subversion/README
A    subversion/autogen.sh
A    subversion/build.conf
…
```

The above command will create a working copy of the latest (unreleased) Subversion source code into a subdirectory named `subversion` in your current working directory. You can adjust that last argument as you see fit. Regardless of what you call the new working copy directory, though, after this operation completes, you will now have the Subversion source code. Of course, you will still need to fetch a few helper libraries (apr, apr-util, etc.)—see the `INSTALL` file in the top level of the working copy for details.

High-speed Tutorial

> “Please make sure your seat backs are in their full, upright position, and that your tray tables are stored. Flight attendants, prepare for take-off….”

What follows is a quick tutorial that walks you through some basic Subversion configuration and operation. When you finish it, you should have a basic understanding of Subversion's typical usage.



The examples used in this appendix assume that you have **svn**, the Subversion command-line client, and **svnadmin**, the administrative tool, ready to go on a Unix-like operating system. (This tutorial also works at the Windows commandline prompt, assuming you make some obvious tweaks.) We also assume you are using Subversion 1.2 or later (run **svn --version** to check.)

Subversion stores all versioned data in a central repository. To begin, create a new repository:

```
$ svnadmin create /path/to/repos
$ ls /path/to/repos
conf/  dav/  db/  format  hooks/  locks/  README.txt
```

This command creates a new directory `/path/to/repos` which contains a Subversion repository. This new directory contains (among other things) a collection of database files. You won't see your versioned files if you peek inside. For more information about repository creation and maintenance, see Chapter 5, *Repository Administration*.

Subversion has no concept of a “project”. The repository is just a virtual versioned filesystem, a large tree that can hold anything you wish. Some administrators prefer to store only one project in a repository, and others prefer to store multiple projects in a repository by placing them into separate directories. The merits of each approach are discussed in the section called “Planning Your Repository Organization”. Either way, the repository only manages files and directories, so it's up to humans to interpret particular directories as “projects”. So while you might see references to projects throughout this book, keep in mind that we're only ever talking about some directory (or collection of directories) in the repository.

In this example, we assume that you already have some sort of project (a collection of files and directories) that you wish to import into your newly created Subversion repository. Begin by or-

ganizing your data into a single directory called `myproject` (or whatever you wish). For reasons that will be clear later (see Chapter 4, *Branching and Merging*), your project's tree structure should contain three top-level directories named `branches`, `tags`, and `trunk`. The `trunk` directory should contain all of your data, while `branches` and `tags` directories are empty:

```
/tmp/myproject/branches/
/tmp/myproject/tags/
/tmp/myproject/trunk/
                     foo.c
                     bar.c
                     Makefile
                     …
```

The `branches`, `tags`, and `trunk` subdirectories aren't actually required by Subversion. They're merely a popular convention that you'll most likely want to use later on.

Once you have your tree of data ready to go, import it into the repository with the **svn import** command (see the section called “Getting Data into your Repository”):

```
$ svn import /tmp/myproject file:///path/to/repos/myproject -m "initial import"
Adding         /tmp/myproject/branches
Adding         /tmp/myproject/tags
Adding         /tmp/myproject/trunk
Adding         /tmp/myproject/trunk/foo.c
Adding         /tmp/myproject/trunk/bar.c
Adding         /tmp/myproject/trunk/Makefile
…
Committed revision 1.
$
```

Now the repository contains this tree of data. As mentioned earlier, you won't see your files by directly peeking into the repository; they're all stored within a database. But the repository's imaginary filesystem now contains a top-level directory named `myproject`, which in turn contains your data.

Note that the original `/tmp/myproject` directory is unchanged; Subversion is unaware of it. (In fact, you can even delete that directory if you wish.) In order to start manipulating repository data, you need to create a new “working copy” of the data, a sort of private workspace. Ask Subversion to “check out” a working copy of the `myproject/trunk` directory in the repository:

```
$ svn checkout file:///path/to/repos/myproject/trunk myproject
A  myproject/foo.c
A  myproject/bar.c
A  myproject/Makefile
…
Checked out revision 1.
```

Now you have a personal copy of part of the repository in a new directory named `myproject`. You can edit the files in your working copy and then commit those changes back into the repository.

- Enter your working copy and edit a file's contents.

- Run **svn diff** to see unified diff output of your changes.
- Run **svn commit** to commit the new version of your file to the repository.
- Run **svn update** to bring your working copy “up-to-date” with the repository.

For a full tour of all the things you can do with your working copy, read Chapter 2, *Basic Usage*.

At this point, you have the option of making your repository available to others over a network. See Chapter 6, *Server Configuration* to learn about the different sorts of server processes available and how to configure them.

Appendix B. Subversion for CVS Users

This appendix is a guide for CVS users new to Subversion. It's essentially a list of differences between the two systems as “viewed from 10,000 feet”. For each section, we provide back-references to relevant chapters when possible.

Although the goal of Subversion is to take over the current and future CVS user base, some new features and design changes were required to fix certain “broken” behaviors that CVS had. This means that, as a CVS user, you may need to break habits—ones that you forgot were odd to begin with.

Revision Numbers Are Different Now

In CVS, revision numbers are per-file. This is because CVS stores its data in RCS files; each file has a corresponding RCS file in the repository, and the repository is roughly laid out according to the structure of your project tree.

In Subversion, the repository looks like a single filesystem. Each commit results in an entirely new filesystem tree; in essence, the repository is an array of trees. Each of these trees is labeled with a single revision number. When someone talks about “revision 54”, they're talking about a particular tree (and indirectly, the way the filesystem looked after the 54th commit).

Technically, it's not valid to talk about “revision 5 of `foo.c`”. Instead, one would say “`foo.c` as it appears in revision 5”. Also, be careful when making assumptions about the evolution of a file. In CVS, revisions 5 and 6 of `foo.c` are always different. In Subversion, it's most likely that `foo.c` did *not* change between revisions 5 and 6.

Similarly, in CVS a tag or branch is an annotation on the file, or on the version information for that individual file, whereas in Subversion a tag or branch is a copy of an entire tree (by convention, into the `/branches` or `/tags` directories that appear at the top level of the repository, beside `/trunk`). In the repository as a whole, many versions of each file may be visible: the latest version on each branch, every tagged version, and of course the latest version on the trunk itself. So, to refine the terms even further, one would often say “`foo.c` as it appears in `/branches/REL1` in revision 5.”

For more details on this topic, see the section called “Revisions”.

Directory Versions

Subversion tracks tree structures, not just file contents. It's one of the biggest reasons Subversion was written to replace CVS.

Here's what this means to you, as a former CVS user:

- The **svn add** and **svn delete** commands work on directories now, just as they work on files. So do **svn copy** and **svn move**. However, these commands do *not* cause any kind of immediate change in the repository. Instead, the working items are simply “scheduled” for addition or deletion. No repository changes happen until you run **svn commit**.
- Directories aren't dumb containers anymore; they have revision numbers like files. (Or more properly, it's correct to talk about “directory `foo/` in revision 5”.)

Let's talk more about that last point. Directory versioning is a hard problem; because we want to allow mixed-revision working copies, there are some limitations on how far we can abuse this model.

From a theoretical point of view, we define “revision 5 of directory `foo`” to mean a specific collection of directory-entries and properties. Now suppose we start adding and removing files from `foo`, and then commit. It would be a lie to say that we still have revision 5 of `foo`. However, if we bumped `foo`'s revision number after the commit, that would be a lie too; there may be other changes to `foo` we haven't yet received, because we haven't updated yet.

Subversion deals with this problem by quietly tracking committed adds and deletes in the `.svn` area. When you eventually run **svn update**, all accounts are settled with the repository, and the directory's new revision number is set correctly. *Therefore, only after an update is it truly safe to say that you have a “perfect” revision of a directory.* Most of the time, your working copy will contain “imperfect” directory revisions.

Similarly, a problem arises if you attempt to commit property changes on a directory. Normally, the commit would bump the working directory's local revision number. But again, that would be a lie, because there may be adds or deletes that the directory doesn't yet have, because no update has happened. *Therefore, you are not allowed to commit property-changes on a directory unless the directory is up-to-date.*

For more discussion about the limitations of directory versioning, see the section called “Mixed Revision Working Copies”.

More Disconnected Operations

In recent years, disk space has become outrageously cheap and abundant, but network bandwidth has not. Therefore, the Subversion working copy has been optimized around the scarcer resource.

The `.svn` administrative directory serves the same purpose as the CVS directory, except that it also stores read-only, “pristine” copies of your files. This allows you to do many things off-line:

svn status
: Shows you any local changes you've made (see the section called “See an overview of your changes”)

svn diff
: Shows you the details of your changes (see the section called “Examine the details of your local modifications”)

svn revert
: Removes your local changes (see the section called “Undoing Working Changes”)

Also, the cached pristine files allow the Subversion client to send differences when committing, which CVS cannot do.

The last subcommand in the list is new; it will not only remove local changes, but it will unschedule operations such as adds and deletes. It's the preferred way to revert a file; running **rm file; svn update** will still work, but it blurs the purpose of updating. And, while we're on this subject…

Distinction Between Status and Update

In Subversion, we've tried to erase a lot of the confusion between the **cvs status** and **cvs update** commands.

The **cvs status** command has two purposes: first, to show the user any local modifications in the working copy, and second, to show the user which files are out-of-date. Unfortunately, because of CVS's hard-to-read status output, many CVS users don't take advantage of this command at all. Instead, they've developed a habit of running **cvs update** or **cvs -n update** to quickly see their changes. If users forget to use the `-n` option, this has the side effect of merging repository changes they may not be ready to deal with.

With Subversion, we've tried to remove this muddle by making the output of **svn status** easy to read for both humans and parsers. Also, **svn update** only prints information about files that are updated, *not* local modifications.

Status

svn status prints all files that have local modifications. By default, the repository is not contacted. While this subcommand accepts a fair number of options, the following are the most commonly used ones:

`-u`
: Contact the repository to determine, and then display, out-of-dateness information.

`-v`
: Show *all* entries under version control.

`-N`
: Run non-recursively (do not descend into subdirectories).

The **status** command has two output formats. In the default “short” format, local modifications look like this:

```
$ svn status
M      foo.c
M      bar/baz.c
```

If you specify the `--show-updates` (`-u`) option, a longer output format is used:

```
$ svn status -u
M            1047   foo.c
       *     1045   faces.html
       *            bloo.png
M            1050   bar/baz.c
Status against revision:   1066
```

In this case, two new columns appear. The second column contains an asterisk if the file or directory is out-of-date. The third column shows the working-copy's revision number of the item. In the example above, the asterisk indicates that `faces.html` would be patched if we updated, and that `bloo.png` is a newly added file in the repository. (The absence of any revision number next to `bloo.png` means that it doesn't yet exist in the working copy.)

At this point, you should take a quick look at the list of all possible status codes in svn status. Here are a few of the more common status codes you'll see:

```
A    Resource is scheduled for Addition
D    Resource is scheduled for Deletion
M    Resource has local Modifications
C    Resource has Conflicts (changes have not been completely merged
       between the repository and working copy version)
X    Resource is eXternal to this working copy (may come from another
       repository).  See the section called “Externals Definitions”
?    Resource is not under version control
!    Resource is missing or incomplete (removed by another tool than
       Subversion)
```

For a more detailed discussion of **svn status**, see the section called “See an overview of your changes”.

Update

svn update updates your working copy, and only prints information about files that it updates.

Subversion has combined the CVS `P` and `U` codes into just `U`. When a merge or conflict occurs, Subversion simply prints `G` or `C`, rather than a whole sentence about it.

For a more detailed discussion of **svn update**, see the section called “Update Your Working Copy”.

Branches and Tags

Subversion doesn't distinguish between filesystem space and “branch” space; branches and tags are ordinary directories within the filesystem. This is probably the single biggest mental hurdle a CVS user will need to climb. Read all about it in Chapter 4, *Branching and Merging*.



Since Subversion treats branches and tags as ordinary directories, always remember to check out the `trunk` (`http://svn.example.com/repos/calc/trunk/`) of your project, and not the project itself (`http://svn.example.com/repos/calc/`). If you make the mistake of checking out the project itself, you'll wind up with a working copy that contains a copy of your project for every branch and tag you have.[1]

Metadata Properties

A new feature of Subversion is that you can attach arbitrary metadata (or “properties”) to files and directories. Properties are arbitrary name/value pairs associated with files and directories in your working copy.

To set or get a property name, use the **svn propset** and **svn propget** subcommands. To list all properties on an object, use **svn proplist**.

For more information, see the section called “Properties”.

Conflict Resolution

[1] That is, providing you don't run out of disk space before your checkout finishes.

CVS marks conflicts with in-line “conflict markers”, and prints a `C` during an update. Historically, this has caused problems, because CVS isn't doing enough. Many users forget about (or don't see) the `C` after it whizzes by on their terminal. They often forget that the conflict-markers are even present, and then accidentally commit files containing conflict-markers.

Subversion solves this problem by making conflicts more tangible. It remembers that a file is in a state of conflict, and won't allow you to commit your changes until you run **svn resolved**. See the section called “Resolve Conflicts (Merging Others' Changes)” for more details.

Binary Files and Translation

In the most general sense, Subversion handles binary files more gracefully than CVS does. Because CVS uses RCS, it can only store successive full copies of a changing binary file. Subversion, however, expresses differences between files using a binary-differencing algorithm, regardless of whether they contain textual or binary data. That means that all files are stored differentially (compressed) in the repository.

CVS users have to mark binary files with `-kb` flags, to prevent data from being garbled (due to keyword expansion and line-ending translations). They sometimes forget to do this.

Subversion takes the more paranoid route—first, it never performs any kind of keyword or line-ending translation unless you explicitly ask it do so (see the section called “Keyword Substitution” and the section called “End-of-Line Character Sequences” for more details). By default, Subversion treats all file data as literal byte strings, and files are always stored in the repository in an untranslated state.

Second, Subversion maintains an internal notion of whether a file is “text” or “binary” data, but this notion is *only* extant in the working copy. During an **svn update**, Subversion will perform contextual merges on locally modified text files, but will not attempt to do so for binary files.

To determine whether a contextual merge is possible, Subversion examines the `svn:mime-type` property. If the file has no `svn:mime-type` property, or has a mime-type that is textual (e.g. `text/*`), Subversion assumes it is text. Otherwise, Subversion assumes the file is binary. Subversion also helps users by running a binary-detection algorithm in the **svn import** and **svn add** commands. These commands will make a good guess and then (possibly) set a binary `svn:mime-type` property on the file being added. (If Subversion guesses wrong, the user can always remove or hand-edit the property.)

Versioned Modules

Unlike CVS, a Subversion working copy is aware that it has checked out a module. That means that if somebody changes the definition of a module (e.g. adds or removes components), then a call to **svn update** will update the working copy appropriately, adding and removing components.

Subversion defines modules as a list of directories within a directory property: see the section called “Externals Definitions”.

Authentication

With CVS's pserver, you are required to “login” to the server before any read or write operation—you sometimes even have to login for anonymous operations. With a Subversion repository using Apache **httpd** or **svnserve** as the server, you don't provide any authentication credentials at the outset—if an operation that you perform requires authentication, the server will

challenge you for your credentials (whether those credentials are username and password, a client certificate, or even both). So if your repository is world-readable, you will not be required to authenticate at all for read operations.

As with CVS, Subversion still caches your credentials on disk (in your `~/.subversion/auth/` directory) unless you tell it not to by using the `--no-auth-cache` option.

The exception to this behavior, however, is in the case of accessing an **svnserve** server over an SSH tunnel, using the `svn+ssh://` URL scheme. In that case, the **ssh** program unconditionally demands authentication just to start the tunnel.

Converting a Repository from CVS to Subversion

Perhaps the most important way to familiarize CVS users with Subversion is to let them continue to work on their projects using the new system. And while that can be somewhat accomplished using a flat import into a Subversion repository of an exported CVS repository, the more thorough solution involves transferring not just the latest snapshot of their data, but all the history behind it as well, from one system to another. This is an extremely difficult problem to solve that involves deducing changesets in the absence of atomicity, and translating between the systems' completely orthogonal branching policies, among other complications. Still, there are a handful of tools claiming to at least partially support the ability to convert existing CVS repositories into Subversion ones.

The most popular (and likely the most mature) conversion tool is cvs2svn (http://cvs2svn.tigris.org/), a Python script originally created by members of Subversion's own development community. This tool is meant to run exactly once: it scans your CVS repository multiple times and attempts to deduce commits, branches, and tags as best it can. When it finishes, the result is a either a Subversion repository or a portable Subversion dumpfile representing your code's history. See the website for detailed instructions and caveats.

Appendix C. WebDAV and Autoversioning

WebDAV is an extension to HTTP, and is growing more and more popular as a standard for file-sharing. Today's operating systems are becoming extremely Web-aware, and many now have built-in support for mounting “shares” exported by WebDAV servers.

If you use Apache as your Subversion network server, then to some extent you are also running a WebDAV server. This appendix gives some background on the nature of this protocol, how Subversion uses it, and how well Subversion interoperates with other software that is WebDAV-aware.

What is WebDAV?

DAV stands for “Distributed Authoring and Versioning”. RFC 2518 defines a set of concepts and accompanying extension methods to HTTP 1.1 that make the web into a more universal read/write medium. The basic idea is that a WebDAV-compliant web server can act like a generic file server; clients can “mount” shared folders over HTTP that behave much like other network filesystems (such as NFS or SMB.)

The tragedy, though, is that despite the acronym, the RFC specification doesn't actually describe any sort of version control. Basic WebDAV clients and servers assume only one version of each file or directory exists, and can be repeatedly overwritten.

Because RFC 2518 left out versioning concepts, another committee was left with the responsibility of writing RFC 3253 a few years later. The new RFC adds versioning concepts to WebDAV, placing the “V” back in “DAV” — hence the term “DeltaV”. WebDAV/DeltaV clients and servers are often called just “DeltaV” programs, since DeltaV implies the existence of basic WebDAV.

The original WebDAV standard has been widely successful. Every modern computer operating system has a general WebDAV client built-in (details to follow), and a number of popular standalone applications are also able to speak WebDAV— Microsoft Office, Dreamweaver, and Photoshop to name a few. On the server end, the Apache webserver has been able to provide WebDAV services since 1998 and is considered the de-facto open-source standard. There are several other commercial WebDAV servers available, including Microsoft's own IIS.

DeltaV, unfortunately, has not been so successful. It's very difficult to find any DeltaV clients or servers. The few that do exist are relatively unknown commercial products, and thus it's very difficult to test interoperability. It's not entirely clear as to why DeltaV has remained stagnant. Some argue that the specification is just too complex, others argue that while WebDAV's features have mass appeal (even the least technical users appreciate network file-sharing), version control features aren't interesting or necessary for most users. Finally, some have argued that DeltaV remains unpopular because there's still no open-source server product which implements it well.

When Subversion was still in its design phase, it seemed like a great idea to use Apache as a network server. It already had a module to provide WebDAV services. DeltaV was a relatively new specification. The hope was that the Subversion server module (**mod_dav_svn**) would eventually evolve into an open-source DeltaV reference implementation. Unfortunately, DeltaV has a very specific versioning model that doesn't quite line up with Subversion's model. Some concepts were mappable, others were not.

What does this mean, then?

First, the Subversion client is not a fully-implemented DeltaV client. It needs certain types of things from the server that DeltaV itself cannot provide, and thus is largely dependent on a number of Subversion-specific HTTP `REPORT` requests that only **mod_dav_svn** understands.

Second, **mod_dav_svn** is not a fully-realized DeltaV server. Many portions of the DeltaV specification were irrelevant to Subversion, and thus left unimplemented.

There is still some debate in the developer community as to whether or not it's worthwhile to remedy either of these situations. It's fairly unrealistic to change Subversion's design to match DeltaV, so there's probably no way the client can ever learn to get everything it needs from a general DeltaV server. On the other hand, **mod_dav_svn** *could* be further developed to implement all of DeltaV, but it's hard to find motivation to do so—there are almost no DeltaV clients to interoperate with.

Autoversioning

While the Subversion client is not a full DeltaV client, nor the Subversion server a full DeltaV server, there's still a glimmer of WebDAV interoperability to be happy about: it's called autoversioning.

Autoversioning is an optional feature defined in the DeltaV standard. A typical DeltaV server will reject an ignorant WebDAV client attempting to do a `PUT` to a file that's under version control. To change a version-controlled file, the server expects a series of proper versioning requests: something like `MKACTIVITY`, `CHECKOUT`, `PUT`, `CHECKIN`. But if the DeltaV server supports autoversioning, then write-requests from basic WebDAV clients are accepted. The server behaves as if the client *had* issued the proper series of versioning requests, performing a commit under the hood. In other words, it allows a DeltaV server to interoperate with ordinary WebDAV clients that don't understand versioning.

Because so many operating systems already have integrated WebDAV clients, the use case for this feature can be incredibly appealing to administrators working with non-technical users: imagine an office of ordinary users running Microsoft Windows or Mac OS. Each user “mounts” the Subversion repository, which appears to be an ordinary network folder. They use the shared folder as they always do: open files, edit them, save them. Meanwhile, the server is automatically versioning everything. Any administrator (or knowledgeable user) can still use a Subversion client to search history and retrieve older versions of data.

This scenario isn't fiction: it's real and it works, as of Subversion 1.2 and later. To activate autoversioning in **mod_dav_svn**, use the `SVNAutoversioning` directive within the `httpd.conf` Location block, like so:

```
<Location /repos>
  DAV svn
  SVNPath /path/to/repository
  SVNAutoversioning on
</Location>
```

When SVNAutoversioning is active, write requests from WebDAV clients result in automatic commits. A generic log message is auto-generated and attached to each revision.

Before activating this feature, however, understand what you're getting into. WebDAV clients tend to do *many* write requests, resulting in a huge number of automatically committed revisions. For example, when saving data, many clients will do a `PUT` of a 0-byte file (as a way of

reserving a name) followed by another PUT with the real file data. The single file-write results in two separate commits. Also consider that many applications auto-save every few minutes, resulting in even more commits.

If you have a post-commit hook program that sends email, you may want to disable email generation either altogether, or on certain sections of the repository; it depends on whether you think the influx of emails will still prove to be valuable notifications or not. Also, a smart post-commit hook program can distinguish between a transaction created via autoversioning and one created through a normal **svn commit**. The trick is to look for a revision property named `svn:autoversioned`. If present, the commit was made by a generic WebDAV client.

Another feature that may be a useful complement for `SVNAutoversioning` comes from Apache's `mod_mime` module. If a WebDAV client adds a new file to the repository, there's no opportunity for the user to set the the `svn:mime-type` property. This might cause the file to appear as generic icon when viewed within a WebDAV shared folder, not having an association with any application. One remedy is to have a sysadmin (or other Subversion-knowledgeable person) check out a working copy and manually set the `svn:mime-type` property on necessary files. But there's potentially no end to such cleanup tasks. Instead, you can use the `ModMimeUsePathInfo` directive in your Subversion `<Location>` block:

```
<Location /repos>
  DAV svn
  SVNPath /path/to/repository
  SVNAutoversioning on

  ModMimeUsePathInfo on

</Location>
```

This directive allows `mod_mime` to attempt automatic deduction of the mime-type on new files that enter the repository via autoversioning. The module looks at the file's named extension and possibly the contents as well; if the file matches some common patterns, then the the file's `svn:mime-type` property will be set automatically.

Client Interoperability

All WebDAV clients fall into one of three categories—standalone applications, file-explorer extensions, or filesystem implementations. These categories broadly define the types of WebDAV functionality available to users. Table C.1, “Common WebDAV Clients” gives our categorization and a quick description of some common pieces of WebDAV-enabled software. More details about these software offerings, as well as their general category, can be found in the sections that follow.

Table C.1. Common WebDAV Clients

Software	Type	Windows	Mac	Linux	Description
Adobe Photoshop	Standalone WebDAV application	X			Image editing software, allowing direct opening from, and writing to, WebDAV URLs
Cadaver	Standalone		X	X	Command-line

Software	Type	Windows	Mac	Linux	Description
	WebDAV application				WebDAV client supporting file transfer, tree, and locking operations
DAV Explorer	Standalone WebDAV application	X	X	X	Java GUI tool for exploring WebDAV shares
Macromedia Dreamweaver	Standalone WebDAV application	X			Web production software able to directly read from and write to WebDAV URLs
Microsoft Office	Standalone WebDAV application	X			Office productivity suite with several components able to directly read from and write to WebDAV URLs
Microsoft Web Folders	File-explorer WebDAV extension	X			GUI file explorer program able to perform tree operations on a WebDAV share
GNOME Nautilus	File-explorer WebDAV extension			X	GUI file explorer able to perform tree operations on a WebDAV share
KDE Konqueror	File-explorer WebDAV extension			X	GUI file explorer able to perform tree operations on a WebDAV share
Mac OS X	WebDAV filesystem implementation		X		Operating system has built-in support for mounting WebDAV shares.
Novell NetDrive	WebDAV filesystem implementation	X			Drive-mapping program for assigning Windows drive letters to a

Software	Type	Windows	Mac	Linux	Description
					mounted remote WebDAV share
SRT WebDrive	WebDAV filesystem implementation	X			File transfer software which, among other things, allows the assignment of Windows drive letters to a mounted remote WebDAV share
davfs2	WebDAV filesystem implementation			X	Linux file system driver that allows you to mount a WebDAV share

Standalone WebDAV applications

A WebDAV application is a program which speakes WebDAV protocols with a WebDAV server. We'll cover some of the most popular programs with this kind of WebDAV support.

Microsoft Office, Dreamweaver, Photoshop

On Windows, there are several well-known applications that contain integrated WebDAV client functionality, such as Microsoft's Office, [1] Adobe's Photoshop, and Macromedia's Dreamweaver programs. They're able to directly open and save to URLs, and tend to make heavy use of WebDAV locks when editing a file.

Note that while many of these programs also exist for the Mac OS X, they do not appear to support WebDAV directly on that platform. In fact, on Mac OS X, the File->Open dialog box doesn't allow one to type a path or URL at all. It's likely that the WebDAV features were deliberately left out of Macintosh versions of these programs, since OS X already provides such excellent low-level filesystem support for WebDAV.

Cadaver, DAV Explorer

Cadaver is a bare-bones Unix commandline program for browsing and changing WebDAV shares. Like the Subversion client, it uses the neon HTTP library—not surprisingly, since both neon and cadaver are written by the same author. Cadaver is free software (GPL license) and is available at http://www.webdav.org/cadaver/.

Using cadaver is similar to using a commandline FTP program, and thus it's extremely useful for basic WebDAV debugging. It can be used to upload or download files in a pinch, and also to examine properties, and to copy, move, lock or unlock files:

```
$ cadaver http://host/repos
```

[1] WebDAV support was removed from Microsoft Access for some reason, but exists in the rest of the Office suite.

```
dav:/repos/> ls
Listing collection `/repos/': succeeded.
Coll: > foobar                                  0  May 10 16:19
      > playwright.el                        2864  May  4 16:18
      > proofbypoem.txt                      1461  May  5 15:09
      > westcoast.jpg                       66737  May  5 15:09

dav:/repos/> put README
Uploading README to `/repos/README':
Progress: [============================>] 100.0% of 357 bytes succeeded.

dav:/repos/> get proofbypoem.txt
Downloading `/repos/proofbypoem.txt' to proofbypoem.txt:
Progress: [============================>] 100.0% of 1461 bytes succeeded.
```

DAV Explorer is another standalone WebDAV client, written in Java. It's under a free Apache-like license and is available at http://www.ics.uci.edu/~webdav/. DAV Explorer does everything cadaver does, but has the advantages of being portable and being a more user-friendly GUI application. It's also one of the first clients to support the new WebDAV Access Control Protocol (RFC 3744).

Of course, DAV Explorer's ACL support is useless in this case, since **mod_dav_svn** doesn't support it. The fact that both Cadaver and DAV Explorer support some limited DeltaV commands isn't particularly useful either, since they don't allow `MKACTIVITY` requests. But it's not relevant anyway; we're assuming all of these clients are operating against an autoversioning repository.

File-explorer WebDAV extensions

Some popular file explorer GUI programs support WebDAV extensions which allow a user to browse a DAV share as if it was just another directory on the local computer, and to perform basic tree editing operations on the items in that share. For example, Windows Explorer is able to browse a WebDAV server as a “network place”. Users can drag files to and from the desktop, or can rename, copy, or delete files in the usual way. But because it's only a feature of the file-explorer, the DAV share isn't visible to ordinary applications. All DAV interaction must happen through the explorer interface.

Microsoft Web Folders

Microsoft was one of the original backers of the WebDAV specification, and first started shipping a client in Windows 98, known as “Web Folders”. This client was also shipped in Windows NT4 and 2000.

The original Web Folders client was an extension to Explorer, the main GUI program used to browse filesystems. It works well enough. In Windows 98, the feature might need to be explicitly installed if Web Folders aren't already visible inside “My Computer”. In Windows 2000, simply add a new “network place”, enter the URL, and the WebDAV share will pop up for browsing.

With the release of Windows XP, Microsoft started shipping a new implementation of Web Folders, known as the “WebDAV mini-redirector”. The new implementation is a filesystem-level client, allowing WebDAV shares to be mounted as drive letters. Unfortunately, this implementation is incredibly buggy. The client usually tries to convert http URLs (`http://host/repos`) into UNC share notation (`\\host\repos`); it also often tries to use Windows Domain authentication to respond to basic-auth HTTP challenges, sending usernames as `HOST\username`. These interoperability problems are severe and documented in numerous places around the web, to the frustration of many users. Even Greg Stein, the original author of Apache's Web-

DAV module, recommends against trying to use XP Web Folders against an Apache server.

It turns out that the original “Explorer-only” Web Folders implementation isn't dead in XP, it's just buried. It's still possible to find it by using this technique:

1. Go to 'Network Places'.
2. Add a new network place.
3. When prompted, enter the URL of the repository, but *include a port number* in the URL. For example, `http://host/repos` would be entered as `http://host:80/repos` instead.
4. Respond to any authentication prompts.

There are a number of other rumored workarounds to the problems, but none of them seem to work on all versions and patchlevels of Windows XP. In our tests, only the previous algorithm seems to work consistently on every system. The general consensus of the WebDAV community is that you should avoid the new Web Folders implementation and use the old one instead, and that if you need a real filesystem-level client for Windows XP, then use a third-party program like WebDrive or NetDrive.

A final tip: if you're attempting to use XP Web Folders, make sure you have the absolute latest version from Microsoft. For example, Microsoft released a bug-fixed version in January 2005, available at http://support.microsoft.com/?kbid=892211. In particular, this release is known to fix a bug whereby browsing a DAV share shows an unexpected infinite recursion.

Nautilus, Konqueror

Nautilus is the official file manager/browser for the GNOME desktop (http://www.gnome.org), and Konqueror is the manager/browser for the KDE desktop (http://www.kde.org). Both of these applications have an explorer-level WebDAV client built-in, and operate just fine against an autoversioning repository.

In GNOME's Nautilus, from the File menu, select Open location and enter the URL. The repository should then be displayed like any other filesystem.

In KDE's Konqueror, you need to use the `webdav://` scheme when entering the URL in the location bar. If you enter an `http://` URL, Konqueror will behave like an ordinary web browser. You'll likely see the generic HTML directory listing produced by **mod_dav_svn**. By entering `webdav://host/repos` instead of `http://host/repos`, Konqueror becomes a WebDAV client and displays the repository as a filesystem.

WebDAV filesystem implementation

The WebDAV filesystem implementation is arguably the best sort of WebDAV client. It's implemented as a low-level filesystem module, typically within the operating system's kernel. This means that the DAV share is mounted like any other network filesystem, similar to mounting an NFS share on Unix, or attaching an SMB share as drive letter in Windows. As a result, this sort of client provides completely transparent read/write WebDAV access to all programs. Applications aren't even aware that WebDAV requests are happening.

WebDrive, NetDrive

Both WebDrive and NetDrive are excellent commercial products which allow a WebDAV share to be attached as drive letters in Windows. We've had nothing but success with these

products. At the time of writing, WebDrive can be purchased from South River Technologies (http://www.southrivertech.com). NetDrive ships with Netware, is free of charge, and can be found by searching the web for “netdrive.exe”. Though it is freely available online, users are required to have a Netware license. (If any of that sounds odd to you, you're not alone. See this page on Novell's website: http://www.novell.com/coolsolutions/qna/999.html)

Mac OS X

Apple's OS X operating system has an integrated filesystem-level WebDAV client. From the Finder, select the Connect to Server item from the Go menu. Enter a WebDAV URL, and it appears as a disk on the desktop, just like any other mounted volume. You can also mount a WebDAV share from the Darwin terminal by using the `webdav` filesystem type with the **mount** command:

```
$ mount -t webdav http://svn.example.com/repos/project /some/mountpoint
$
```

Note that if your **mod_dav_svn** is older than version 1.2, OS X will refuse to mount the share as read-write; it will appear as read-only. This is because OS X insists on locking support for read-write shares, and the ability to lock files first appeared in Subversion 1.2.

One more word of warning: OS X's WebDAV client can sometimes be overly sensitive to HTTP redirects. If OS X is unable to mount the repository at all, you may need to enable the BrowserMatch directive in the Apache server's `httpd.conf`:

```
BrowserMatch "^WebDAVFS/1.[012]" redirect-carefully
```

Linux davfs2

Linux davfs2 is a filesystem module for the Linux kernel, whose development is located at http://dav.sourceforge.net/. Once installed, a WebDAV network share can be mounted with the usual Linux mount command:

```
$ mount.davfs http://host/repos /mnt/dav
```

Appendix D. Third Party Tools

Subversion's modular design (covered in the section called “Layered Library Design”) and the availability of language bindings (as described in the section called “Using Languages Other than C and C++”) make it a likely candidate for use as an extension or backend to other pieces of software. For a listing of many third-party tools that are using Subversion functionality under-the-hood, check out the Links page on the Subversion website (http://subversion.tigris.org/project_links.html).

Appendix E. Copyright

Copyright (c) 2002-2007
Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato.

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit http://creativecommons.org/licenses/by/2.0/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

A summary of the license is given below, followed by the full legal text.

You are free:

* to copy, distribute, display, and perform the work
* to make derivative works
* to make commercial use of the work

Under the following conditions:

Attribution. You must give the original author credit.

* For any reuse or distribution, you must make clear to others the license terms of this work.

* Any of these conditions can be waived if you get permission from the author.

Your fair use and other rights are in no way affected by the above.

The above is a summary of the full license below.

==========

Creative Commons Legal Code
Attribution 2.0

CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO WARRANTIES REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS LIABILITY FOR DAMAGES RESULTING FROM ITS USE.

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

   a. "Collective Work" means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with a number of other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.

   b. "Derivative Work" means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License.

   c. "Licensor" means the individual or entity that offers the Work under the terms of this License.

   d. "Original Author" means the individual or entity who created the Work.

   e. "Work" means the copyrightable work of authorship offered under the terms of this License.

   f. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

2. Fair Use Rights. Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

   a. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works;

   b. to create and reproduce Derivative Works;

   c. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;

   d. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission Derivative Works.

e.

For the avoidance of doubt, where the work is a musical composition:

i. Performance Royalties Under Blanket Licenses. Licensor waives the exclusive right to collect, whether individually or via a performance rights society (e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work.

ii. Mechanical Rights and Statutory Royalties. Licensor waives the exclusive right to collect, whether individually or via a music rights agency or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).

f. Webcasting Rights and Statutory Royalties. For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions).

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. Restrictions.The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any reference to such Licensor or the Original Author, as requested. If You create a Derivative Work, upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work any reference to such Licensor or the Original Author, as requested.

b. If you distribute, publicly display, publicly perform, or publicly digitally perform the Work or any Derivative Works or

Collective Works, You must keep intact all copyright notices for the Work and give the Original Author credit reasonable to the medium or means You are utilizing by conveying the name (or pseudonym if applicable) of the Original Author if supplied; the title of the Work if supplied; to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and in the case of a Derivative Work, a credit identifying the use of the Work in the Derivative Work (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Derivative Work or Collective Work, at a minimum such credit will appear where any other comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

   a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Derivative Works or Collective Works from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

   b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

   a. Each time You distribute or publicly digitally perform the Work or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

b. Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, neither party will use the trademark "Creative Commons" or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons' then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time.

Creative Commons may be contacted at http://creativecommons.org/.

==========

Index

B

BASE, 38

C

COMMITTED, 38
Concurrent Versions System (CVS), xiii

H

HEAD, 38

P

PREV, 38
properties, 40

R

repository
  hooks
    post-commit, 317
    post-lock, 321
    post-revprop-change, 319
    post-unlock, 323
    pre-commit, 316
    pre-lock, 320
    pre-revprop-change, 318
    pre-unlock, 322
    start-commit, 315
revisions
  revision keywords, 38
  specified as dates, 39

S

Subversion
  history of, xix
svn
  subcommands
    add, 205
    blame, 207
    cat, 208
    checkout, 210
    cleanup, 212
    commit, 213
    copy, 215
    delete, 218
    diff, 220
    export, 224
    help, 226
    import, 227
    info, 229
    list, 232
    lock, 234
    log, 236
    merge, 240
    mkdir, 242
    move, 244
    propdel, 246
    propedit, 247
    propget, 248
    proplist, 250
    propset, 252
    resolved, 254
    revert, 256
    status, 258
    switch, 262
    unlock, 265
    update, 267
svnadmin
  subcommands
    create, 270
    deltify, 271
    dump, 272
    help, 274
    hotcopy, 275
    list-dblogs, 276
    list-unused-dblogs, 277
    load, 278
    lslocks, 279
    lstxns, 280
    recover, 281
    rmlocks, 283
    rmtxns, 284
    setlog, 285
    verify, 286
svnlook
  subcommands
    author, 288
    cat, 289
    changed, 290
    date, 291
    diff, 292
    dirs-changed, 293
    help, 294
    history, 295
    info, 296
    lock, 297
    log, 298
    propget, 299
    proplist, 300
    tree, 301
    uuid, 302
    youngest, 303
svnsync
  subcommands
    copy-revprops, 305
    initialize, 306
    synchronize, 307
svnversion, 310